Peter Rugg

Parallel Codebreaking

Computer Science Tripos – Part II

Churchill College

18th May, 2018

Proforma

Name:	**Peter Rugg**
College:	**Churchill College**
Project Title:	**Parallel Codebreaking**
Examination:	**Computer Science Tripos – Part II, July 2018**
Word Count:	**11,984**[1]
Project Originator:	Prof. Simon Moore
Supervisor:	Prof. Simon Moore

Original Aims of the Project

The project aimed to investigate the algorithm used to break the German World War II Lorenz SZ42 cipher and produce an efficient implementation. The key area of interest was to explore the parallelism that exists in the algorithm, exploiting it to produce a multi-threaded implementation, a GPU implementation, and (as an extension) an FPGA implementation, and compare their performance.

Work Completed

The single-threaded CPU, multi-threaded CPU, GPU, and FPGA versions of the code-breaking algorithm were all implemented, changing the algorithm significantly at multiple levels to reveal parallelism suitable for each architecture. Additionally, a vectorised version was produced using AVX extensions. Thorough evaluation was performed over a range of texts with different properties, comparing the implementations' speed and probability of finding the correct key against each other and a baseline Ada version.

Special Difficulties

None.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Peter Rugg of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Contents

List of Figures

For more information on each figure see Appendix A.

Acknowledgements

Many thanks to:

- my supervisor, Prof. Simon Moore for his helpful guidance and feedback;
- my Director of Studies, Dr John Fawcett, for his advice and support;
- Dr Theo Markettos for guidance on setting up Linux on the DE1-SoC board;
- friends and family for proofreading.

Chapter 1

Introduction

This project aims to improve the efficiency of breaking the Lorenz SZ42 cipher through efficient implementation and extraction of parallelism. The project has been a success, with a four-times speedup in single-threaded performance over the starting point Ada code [10]. The multi-threaded implementation achieves a further two-times speedup (on a four-core machine), as opposed to the GPU implementation's four-times speedup over the single-threaded C code, and the FPGA implementation's 25-times speedup. All implementations are significantly more reliable than the original Ada code.

During the second half of the twentieth century, advances in fabrication technology allowed transistor density and clock speeds to grow exponentially [6]. This led to rapidly improving single-threaded performance: the same code could be run on a newer machine and perform significantly better. However, physical limitations – most notably power dissipation – have prevented clock speeds from increasing further. Thus, the emphasis in fast computation has moved towards exploiting parallelism to achieve increased performance via concurrency. Cryptography is a fruitful area for this approach, since many algorithms for breaking ciphers require iterating over a large number of possible keys. Weaknesses in ciphers present an opportunity to reduce the number of keys to be tried as more intelligent approaches allow algorithms to reduce the search space. However, the coordination required to determine which keys to try makes parallelisation more difficult. This introduces an interesting tradeoff between executing a large amount of work in parallel and determining the most promising keys to try.

The Lorenz Cipher SZ42 machine (Figure 1.1) was used during World War II by the Germans for high-priority communications. However, codebreakers[1] at Bletchley Park were able to determine the operation of the machine without ever seeing one [5]. Furthermore, a reliable, fast algorithm was developed for determining the key settings based only on the ciphertext and light assumptions about the distribution of characters in the

[1]Technically, codes are word-level substitutions whereas Lorenz is a cipher: combining each character mathematically with a key. However, throughout the dissertation I will refer to the process as codebreaking as does Gannon [4].

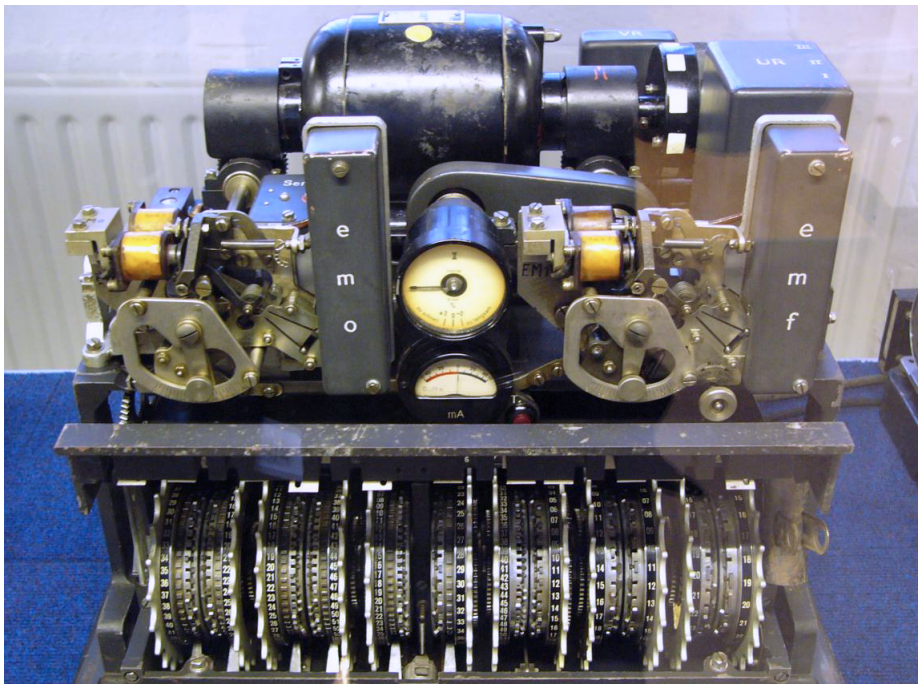

Figure 1.1: A Lorenz SZ42 machine. Image taken from `https://en.wikipedia.org/wiki/File:Lorenz-SZ42-2.jpg`.

plaintext. This project's aim is to investigate the different approaches that could be taken to implement this algorithm with today's technology.

The machine uses a set of wheels rotating with coprime periods to generate a bit stream with an enormous interval (approximately 1.6×10^{19} characters) between repeats. This bit stream is XOR'ed with the input characters to encrypt them. The algorithm to break the key – the starting position of each of the wheels – relies on weaknesses in the design that allow statistical properties of the plaintext to become visible when the ciphertext is decrypted with a correct partial key [5]. By performing frequency analysis on these partially decrypted texts, guesses for each part of the key can be scored for likelihood, allowing the total key to be built up gradually. This is the work that was automated by the Colossus machines. This project considers alternative methods of breaking up the key and generating guesses in some detail, since different options offer different trade-offs between total decrypts required and available parallelism.

This project builds on the work of Schüth [10], in turn building on the work of Bletchley Park codebreakers [5]. Whilst of historical interest, the work is not directly applicable to breaking modern encryption, which requires a great deal more effort to break than Lorenz due to being designed to resist methods of attack such as those used in this project. However, determining how to parallelise algorithms is currently an active area of research in Computer Science.

Throughout the project, many different approaches to parallelisation were explored, which can broadly be split into three categories:

- Instruction-Level Parallelism (ILP): individual instructions that can be executed in parallel. In modern CPUs, this is extracted dynamically in hardware by techniques such as pipelining and superscalar execution. This is present in the encryption process to a large extent, e.g. the different bits of the key can be determined independently.
- Thread-Level Parallelism (TLP): high-level tasks that can be executed concurrently. This is exploited in Multiple Instruction, Multiple Data (MIMD) CPUs by multithreading. The codebreaking algorithm contains TLP in the process of scoring different guesses of the key.
- Data-Level Parallelism (DLP): the same operations being executed on multiple pieces of data that do not depend on each other. DLP can be exploited by CPU vector instructions, and at a larger scale by GPUs. To some extent this can be exploited during encryption, e.g. in the updating of the wheels of the SZ42 machine. DLP is also present at a higher level in the scoring of different keys when carefully re-expressed, although this overlaps with the TLP mentioned above.

By designing custom hardware, we can extract parallelism at multiple levels of an algorithm, since many components can operate in parallel in a circuit, potentially implementing the behaviour of many CPU instructions in a single clock cycle. This essentially extracts as much ILP as possible from the application. DLP and TLP can be extracted in the same way as in a CPU or GPU if required, since components can be duplicated. However, the ability to exploit additional parallelism comes at the cost of increased design effort. Field-Programmable Gate Arrays (FPGAs) are a type of reconfigurable hardware, allowing rapid prototyping of hardware designs. In order to support reconfigurability, FPGAs operate at a significantly lower clock speed (hundreds of Megahertz) than traditional CPUs. Application-Specific Integrated Circuits (ASICs) can be synthesised if a higher clock speed is required, but the cost of this process is prohibitive for this project.

Chapter 2

Preparation

This chapter describes the work done before starting implementation. This mainly consisted of researching the Lorenz cipher and the algorithm to break it, from Schüth's code [10]. This in turn required me to learn Ada – the source language – to the point where I could understand the (uncommented) code and produce my own annotations to explain the algorithm. This chapter describes the cipher and the process used to break it in some detail since this background is required to understand the decisions discussed in the Implementation. In §§2.4-2.6, I describe the software (and hardware) engineering principles employed and tools used to ensure the project ran smoothly.

2.1 Lorenz cipher

The Lorenz SZ42 machine implements a stream cipher: input characters are concealed by an XOR operation with a pseudo-random stream of bits. If the bits were perfectly unpredictable (i.e. a one-time pad), the cipher would achieve perfect secrecy. However, limitations on the amount of key information that could be distributed to Lorenz operators and the limited state within the machine forced an approximation to this randomness to be used. The approximation takes the form of the machine extrapolating a pseudo-random pad from the key by using it to set the initial conditions of mechanical wheels within the machine, which then rotate deterministically to produce a stream of bits. The wheels are designed so that the period (number of characters encrypted until the stream repeats) is long enough (approximately 1.6×10^{19} characters) that the key repeating is not a concern.

Since breaking the cipher relies on the details of the generation of the pseudo-random stream, we must look in more detail at the exact workings of the machine. The i'th input character, I^i, is represented as five bits using the International Telegraph Alphabet No. 2 (Appendix B). There are twelve wheels in the machine. Each wheel has a fixed period, all

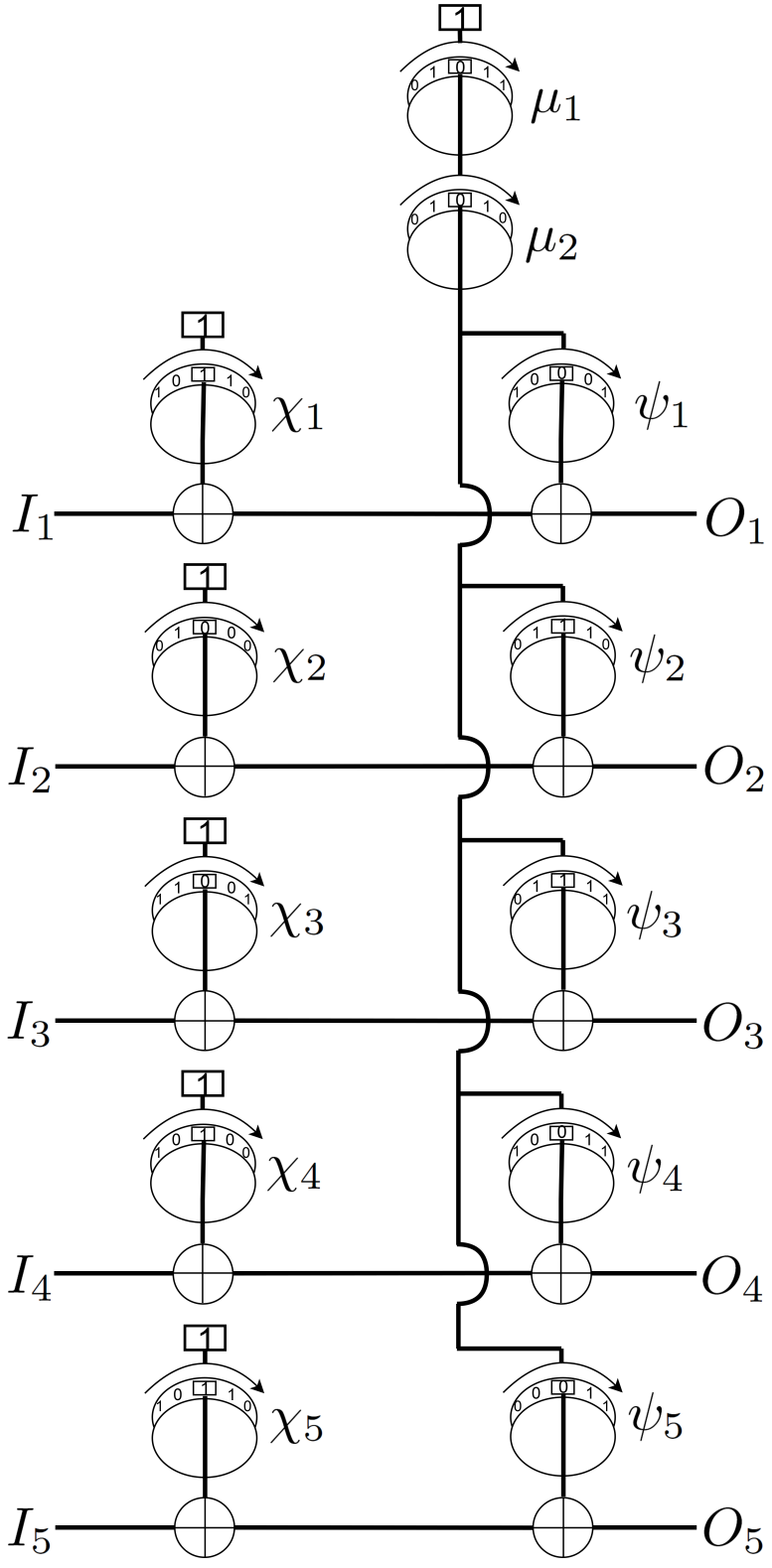

Figure 2.1: The mechanism of the SZ42 machine. Each input bit is XOR'ed with the bits in the active position of the corresponding χ and ψ wheels. The χ wheels and μ_1 wheel rotate on every character whereas the ψ wheels rotate only if the active position of the μ_2 wheel is a 1. The μ_2 wheel in turn rotates only if the μ_1 wheel has a 1 in its active position.

of which are coprime to maximise the interval before the key stream repeats[1]. A wheel's initial state can be altered in two ways: the *pattern* of the wheel refers to whether each position has its "cam" set (i.e. whether it will produce a 1 or a 0); the *position* refers to the starting orientation of the wheel. The wheels were divided by codebreakers at Bletchley Park into three groups, designated χ, μ, and ψ. Figure 2.1 shows the mechanism of the machine.

Wheel	χ_1	χ_2	χ_3	χ_4	χ_5	μ_1	μ_2	ψ_1	ψ_2	ψ_3	ψ_4	ψ_5
Period	41	31	29	26	23	61	37	43	47	51	53	59

Figure 2.2: The periods of wheels in the SZ42 machine

Labelling the χ and ψ wheels 1-5, the i'th output character, O^i, can therefore be described as:

$$O_n^i = I_n^i \oplus \chi_n^i \oplus \psi_n^i \qquad n \in \{1,2,3,4,5\} \text{ denotes bit-select} \tag{2.1}$$

Since the only interaction between the wheels and the input is via XOR'ing, repeating the encryption process with the same initial key configuration on the output text recovers the original text. Hence, encryption and decryption are identical processes.

Additional complications, called *limitations*, were included in the SZ42 machines. These add a dependence on the state of the χ wheels to determine whether the ψ wheels advance. One example – the χ-2 *limitation* – advances the ψ wheels if the active position of the μ_2 wheel contains a 1 (as normal) or the previous position of the χ_2 wheel contained a 0, or both.

2.2 Codebreaking approach

Breaking a Lorenz cipher key requires determining the patterns (whether each wheel position yields 1 or 0) and the wheels' initial orientations. The first step was performed using "depths": ciphertexts known to correspond to very similar plaintexts, encrypted with the same key due to operator error and intercepted. The patterns would be changed by the Germans relatively infrequently (around once per month). Schüth's code and this project only consider the determination of the initial orientation of the wheels given the patterns, which can be performed as a ciphertext-only attack.

During a Cipher Event at Bletchley Park in 2007, participants were asked to write software to break the wheel positions of Lorenz ciphertexts. Joachim Schüth, the winner of the competition, released his code [10]: a large part of the preparation consisted of studying this code to understand how the codebreaking process works. This was made

[1]Since not all wheels rotate every character, the machine does not visit every possible state, and so the total length before the key stream repeats is actually shorter than the product of the wheel periods.

difficult by the lack of comments in the code, and my inexperience with Ada (the source language).

2.2.1 High-level approach

The key to the Lorenz cipher can be broken in three phases [5]: first the χ wheel starting positions are determined, then the μ positions, and finally the ψ positions. Weaknesses in the cipher allow these phases to be carried out without dependence on future stages. This reduces the effective key-space size from the product of all of the wheel periods (1.60×10^{19}) to the sum over the three sets of wheels – χ, μ, ψ – of the products of their periods (3.44×10^{8}).

The main vulnerability in the cipher arises from the complexity in advancing the ψ wheels. Assuming a uniform pattern on the μ wheels, since the ψ wheels advance only when the active position of the μ_2 wheel is a 1, the ψ wheels only advance on average half of the time. The other half, the ψ component of the key is unchanged between consecutive characters. Hence, taking the XOR of adjacent encrypted characters (a technique known as *differencing*) partially removes the ψ component of the key:

$$\begin{aligned} O_n^i \oplus O_n^{i-1} &= (I_n^i \oplus \chi_n^i \oplus \psi_n^i) \oplus (I_n^{i-1} \oplus \chi_n^{i-1} \oplus \psi_n^{i-1}) && \text{by (2.1)} \\ &= (I_n^i \oplus I_n^{i-1}) \oplus (\chi_n^i \oplus \chi_n^{i-1}) \oplus (\psi_n^i \oplus \psi_n^{i-1}) && \oplus \text{ is associative and commutative} \\ &= (I_n^i \oplus I_n^{i-1}) \oplus (\chi_n^i \oplus \chi_n^{i-1}) && \text{in steps where } \psi \text{ did not advance} \end{aligned}$$

Therefore if the position of the n'th χ wheel is guessed correctly, then $I_n^i \oplus I_n^{i-1}$ can be recovered. By computing this for a length of ciphertext, any statistical properties of the XOR between adjacent characters in the plaintext can be observed. This can be used to score χ wheel settings, i.e. determine which are more likely. Pearson's chi-squared[2] test statistic (see §2.2.2) is a suitable metric: higher scores are taken to correspond to a "better" key.

[2]To avoid confusion between χ wheels and the chi-squared test, the former context will always be denoted χ and the latter chi.

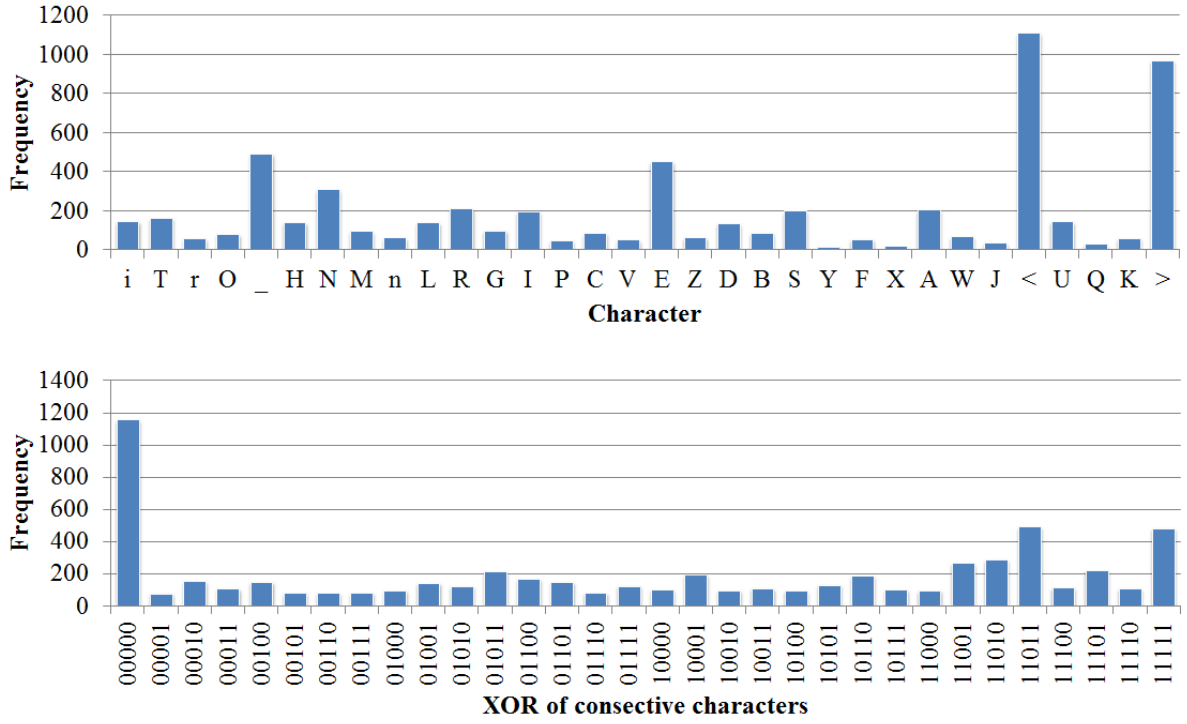

Figure 2.3: Distribution of a plaintext from the codebreaking competition, showing the frequencies of characters and of XORs between adjacent characters. See Appendix B for the meanings of the characters. Note the huge frequency of 00000 (i.e. the same character repeated) in the *differenced* characters, arising from the doubling of gratuitous shift characters (< and >). Implementations were tested and evaluated on these and much harder (more uniform) texts.

This vulnerability decouples the determination of the χ starting positions from those of the other wheels. However, the plaintext's non-uniformity in individual bits can be very small, so the χ wheels still need to be considered together to detect meaningful correlation. The search space size is the product of the periods of the χ wheels (22,041,682) meaning a brute-force search is just about feasible. However, as discussed in §2.2.3, probabilistic searches can be carried out much more quickly, at the expense of a chance of failing to find the optimal positions. Note also that, at this point, it is not known on which characters the ψ wheels advance. Assuming that when the ψ wheels do advance their difference is uniform, roughly half of the character differences are corrupted by the ψ-difference, reducing the observed non-uniformity of the decrypted text. Since this affects all scores the same, chi-squared scores can still be used to compare candidate χ settings, but with additional noise reducing the distinction between correct and incorrect settings. Limitations amplify this effect since they cause the ψ wheels to advance more frequently. However, I augmented the algorithm (§3.1.1) by taking the *limitation* into account during the χ-setting to make it easier to find the correct χ_2 setting when a *limitation* is used.

Once the χ starting positions have been found, the μ starting positions can be in-

vestigated. As noted above, the chi-squared score of a text is reduced by pollution from the steps when the ψ wheels advanced. The μ settings determine on which steps the ψ wheels advance, so candidate μ settings can be scored by determining the chi-squared score of *differenced* texts decrypted with the χ settings found previously but ignoring the characters where the candidate μ settings would lead to the ψ wheels advancing. Due to the heavy interdependence between the two μ wheels, the probabilistic methods used on the χ and ψ wheels do not work well. Fortunately, the search space is small (2,257 possibilities) so a brute-force search is ideal.

Finally, the ψ settings can be determined. Candidate ψ settings are scored by the non-uniformity of the non-*differenced* text when decrypted with the known χ and μ settings and postulated ψ settings. Again, this lends itself to a probabilistic local search.

2.2.2 Scoring keys

A metric is required to judge how promising a key is, based on the properties of the text after decryption with the key. For this purpose, Schüth uses Pearson's chi-squared test. This defines a test-statistic as follows:

$$\text{chi-squared} = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{2.2}$$

where n is the number of possible outcomes, O_i is the number of observations of outcome i actually seen, and E_i is the number of observations of outcome i expected under the null hypothesis.

We apply the chi-squared test to:

- the decrypted stream of characters (for the ψ-setting);
- the differences (XOR) between adjacent decrypted characters (for the χ-setting and μ-setting).

In both cases, $n = 32$ is the number of possible characters or character differences, since characters are five bits. O_i is the number of characters in the decrypted stream taking value i. E_i is the number of characters we would expect to take the value i were the text uniform, i.e. the length of the text divided by n.

The chi-squared statistic can be used to determine the significance of a deviation from the expected outcome proportions, allowing a decision to be made about the likelihood that the data was indeed produced by the expected distribution. Since natural languages have biases in the frequencies of characters, and combinations of adjacent characters [3], a correct key is likely to give a less uniform sequence. We only need to measure the relative fitness of the candidate keys, and so the chi-squared statistic can be used for comparison directly rather than computing the significance against the chi-squared distribution.

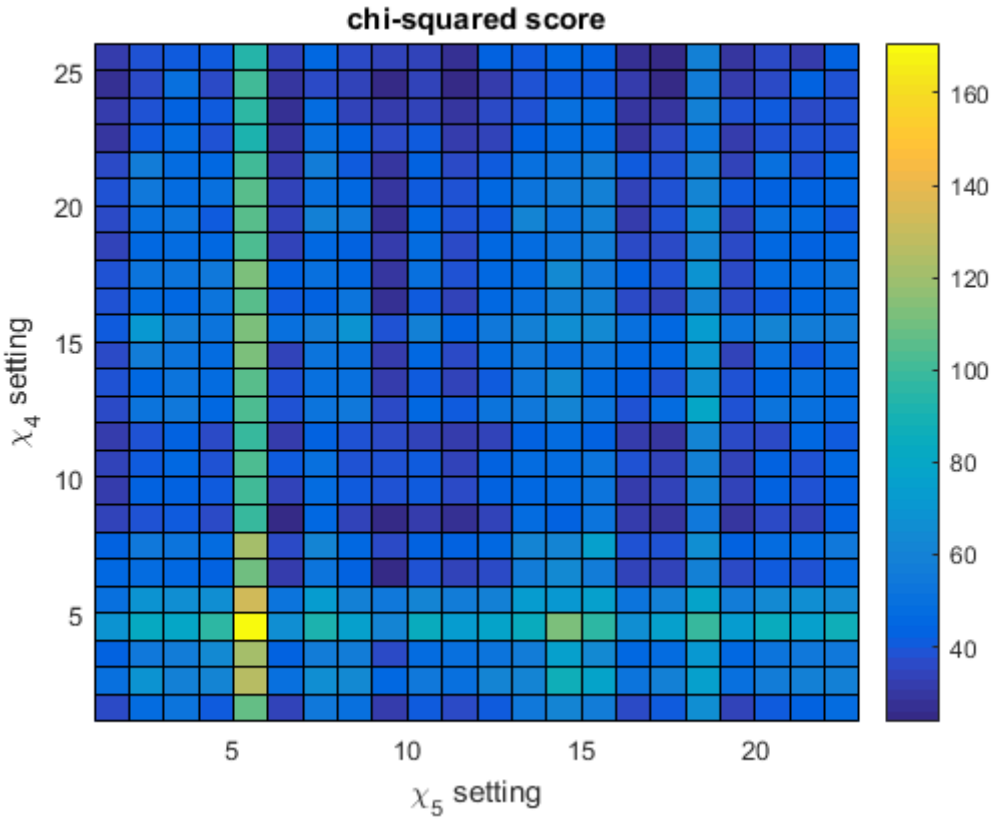

Figure 2.4: Chi-squared scores for all χ_4 and χ_5 settings. Correct setting is $\chi_4 = 4, \chi_5 = 5$. The χ_2 wheel has been set to the correct position, making the correlation more pronounced. The other wheels have been set randomly.

If the distribution of the plaintext characters were known, more accurate scoring could be performed by comparing with the expected character frequencies. However, this project aims to determine the key with minimal assumptions on the plaintext, so only the deviation of the text from uniform will be exploited.

2.2.3 Probabilistic search

A probabilistic search method can greatly improve the speed of finding the settings of the χ and ψ wheels compared to a brute-force search. The aim is to find the global maximum of the fitness function (i.e. chi-squared score of ciphertext decrypted with candidate key) over all keys. The search space in the cases of both χ and ψ is five-dimensional: the starting position for each wheel being a dimension. The key observation is that, in each dimension, the correct position is slightly more likely to give a higher chi-squared score than the other positions. This effect is magnified when the settings for other wheels are also correct, as then the non-uniformity of the two wheels can be correlated to produce greater non-uniformity. To illustrate, Figure 2.4 shows the chi-squared scores for all possible settings of the last two χ wheels.

```
Input: wheel_positions, trial_wheels, timeout
randomise positions of trial_wheels;
unchanged_count = 0;
while unchanged_count < timeout do
    wheel = random from trial_wheels;
    wheel_positions[wheel] = random from 1..wheel_periods[wheel];
    find score of wheel_positions;
    if score has improved then
        unchanged_count = 0;
    else
        revert wheel_positions;
        ++unchanged_count;
    end
end
```

Figure 2.5: Pseudocode for Schüth's Monte Carlo search technique

Schüth's approach to this search is implemented in a routine called `Carlo_Setting`, presumably referring to the Monte Carlo class of random algorithms. As shown in Figure 2.5, the idea is to randomly permute a random wheel, but revert if this does not improve the score. The process is repeated until a certain timeout (which Schüth sets as 1,000) passes without an improvement being found.

This procedure only finds the globally maximal setting with some probability $p < 1$, a property of the non-uniformity and length of the plaintext. However, we can repeat the process n times with different random starting positions, taking the result with the best score. This improves the probability of finding the maximally scoring setting to $1 - (1 - p)^n$, since each run is an independent Bernoulli trial. Schüth sets $n = 10$.

Throughout the project, I investigated multiple alternative approaches to this search. See §3.1.1 for a more formal discussion of this and an alternative algorithm and §4.3 for a comparison of their effectiveness.

2.3 Starting point

The starting point is as stated in Proposal (Appendix D). In short, Schüth's Ada code [10] was used as a starting point for the algorithm. My relevant experience beyond the Tripos comes from an internship working in C and an internship working with Hardware Design Languages.

2.4 Project management

The project implementation is naturally divided into four components: the C, multi-threaded, GPU, and FPGA implementations. The latter three each rely on the C implementation, which fortunately poses the lowest risk of the four, since there is existing Ada code [10] to work from and compare outputs with. However, once the C implementation was complete, there was no interdependence between the higher risk GPU and FPGA implementations. The priority was therefore to implement the C version as quickly as possible, to identify early any unforeseen aspects of the algorithm making parallelisation more challenging. Also, the CPU application was thoroughly tested (§2.5) before proceeding to the parallel implementations. The FPGA extension was not undertaken until it was clear the other versions could be completed in time, potentially allowing more time to be devoted to the core parts of the project if required.

Requirements analysis is key to provide criteria as to whether the project has been a success, and thus focus development on the critical features. The functional goal of each implementation of the program is clear: it should be given an encrypted text and the wheel patterns that were used to encrypt it, and determine the initial positions of the wheels that were used to encrypt it. Beyond this, the implementation should run as fast as possible. However, the probabilistic nature of the process makes functional success difficult to define, since it is impossible to guarantee that the key output is always correct for all texts (since multiple plaintexts could encrypt with different keys to give the same ciphertext). The aim taken was to work on texts of around the same length as those in the Bletchley Park competition [10], i.e. around 6,000 characters, and not to rely on a known plaintext distribution in the breaking process. The implementations should be at least as reliable as the Ada code. However, due to the statistical properties of the competition texts (see Appendix C), the Ada code finds the correct key with a very high probability, making it difficult to compare with an alternative implementation. Therefore the code should be at least as reliable as the Ada implementation on more difficult texts, designed during evaluation.

The backup procedures listed in the Proposal were followed without issue.

2.5 Tools

This section lists the main tools used during the project, along with brief justification of their selection.

The main languages used were:

- C: used for baseline implementation. Since the focus in the project is on efficiency, it was important to use a language where operation costs are minimal and predictable.

Also, C allowed the CUDA implementation to be written as an extension from the existing C code, making development easier and allowing more meaningful performance comparison, since the language overheads are the same. I also used C for the software to coordinate the FPGA implementation.

- CUDA: used for GPU implementation. This was chosen over OpenCL (the main alternative) due to better performance [7], especially on NVIDIA hardware.
- SystemVerilog: used for FPGA implementation. SystemVerilog provides richer types and static checks, e.g. `always_comb` and `always_ff` blocks, than Verilog. However, SystemVerilog gives a clearer idea of what programming constructs map to in hardware than higher level languages like Chisel and Bluespec. Again, this makes it easier to determine the performance impact of different choices.

Note that I used C++ to a very small extent to use the `<random>` header's Marsenne Twister implementation instead of the C `rand()` function, which provides limited randomness.

The project relied on the following tools:

- GCC: used to compile the baseline C version of the algorithm.
- Valgrind: used for C memory error checking.
- GProf: used to profile C applications to determine optimisation strategy.
- POSIX Threads (pthreads): library used for C multi-threading.
- CLion: IDE providing C debugging tools.
- GNAT and GNAT Programming Studio: used to compile Schüth's Ada code.
- Visual Studio 2015: used to develop GPU implementation due to support for CUDA.
- CUDA Visual Profiling tool: used to profile the GPU implementation.
- ModelSim: used to simulate hardware designs to determine functional correctness.
- Quartus: used to synthesise hardware designs for FPGA.
- ECAD Labs Virtual Machine: used as a preconfigured environment for ModelSim and Quartus.
- ECAD Labs DE1-SoC board: used for FPGA and Hard Processor System to test and evaluate hardware design.
- Altera Soc Embedded Development Suite: used to compile software for the DE1-SoC.

- Linux Board Support Package for DE1-SoC [12]: included Linux boot image and setup instructions.
- git: used for version control.

2.5.1 Licensing

The above tools were all used under educational, not-for-profit, or GPL licenses, making my use fully legitimate.

The texts used for evaluation (Appendix C) contained licenses allowing free copy and use provided the license was also copied: a copy of these licenses was made in the code repository.

No part of Schüth's actual code is used in my own, only the algorithms described within it: these are not protected by Copyright law therefore there is no legal issue. Schüth released his source code since "others might find the algorithms interesting" [10], so my use also falls within the spirit of his release of the code.

2.6 Testing strategy

Testing is important at every stage of development to ensure time is not wasted building on flawed work, and that results gathered for evaluation are meaningful. Therefore, I produced unit tests for each aspect of the code as it was written. In addition, the nature of the project allows for definitive automated overall testing of each aspect:

- I tested the baseline C version by comparing the output on randomly generated texts and keys with Schüth's code. To verify codebreaking, I checked that the key found by the algorithm matched the known correct key[3]. Since the algorithm is probabilistic, the percentage of correctly decrypted texts was compared to Schüth's Ada code to check that bugs were not reducing success probability.
- I tested the GPU version similarly. Testing the extent to which GPU parallelism was exploited relied on profiling the code with CUDA's Visual Profiler (§4.5).
- Initially, I performed functional testing on each module of the FPGA version in simulation. Only when the design was reliably producing correct results did I move onto synthesising it, at which point the outputs on a large volume of random plaintexts could again be compared to known results.

[3]Note that it is possible if the wheel patterns have certain symmetries for there to be multiple correct keys, so I had to take this into account in testing and evaluation.

Chapter 3

Implementation

This chapter describes the work undertaken to produce the different versions of the algorithm, discussing the interesting challenges and trade-offs considered. While implementing the initial single-threaded C baseline version of the code, it became clear that several changes could be made – both at a low-level and algorithmically – to improve speed and reliability. I next augmented this baseline version with parallelism in two ways: vectorising the code manually with Intel's Advanced Vector eXtensions (AVX), and multi-threading using pthreads. The GPU version was implemented in CUDA, which I learned from scratch for the project. Significant algorithmic changes were also required to exploit GPU parallelism fully. Finally, I designed a memory-mapped peripheral FPGA device capable of the complete process of applying and scoring a key. I paid particular attention to keeping the design low-area, for instance by implementing the wheels as circular shift registers rather than arrays, and adapting the scoring process to reduce the arithmetic required. This increased the parallelism exploitable by duplicating codebreaking devices on the same FPGA. I also implemented the C software to coordinate the breaking process from a Hard Processor System (HPS).

3.1 Baseline implementation

The first step was to implement a non-parallel C baseline version of Schüth's Ada code. The basis implementation should be as easy as possible to adapt to the later versions: multi-threaded, GPU, and FPGA. This led to some interesting design decisions, which will be discussed in this section. The foremost of these was to change the local search algorithm used for parts of the codebreaking process. Also, I had to make various lower-level decisions about representation of data to suit the target architectures.

3.1.1 Algorithm

Schüth's algorithm includes a Monte Carlo-based search phase (§2.2.3). This chooses a random wheel, sets it to a random position, evaluates the new chi-squared score, then keeps the new wheel setting if the score has improved. This process is repeated until there is no improvement for some fixed number of attempts, which Schüth sets as 1,000.

Formally, here we are using a "local search" algorithm, since in the cases where this step is applied – to the χ and ψ wheels – searching the entire space (3.2×10^8 and 2.2×10^7 keys respectively) is slow. We instead define a neighbouring relation: two configurations are neighbours if they differ in the position of exactly one wheel. The aim is to find the global maximum score by "hill climbing": following neighbours according to some heuristic. In this terminology, Schüth's approach is "first choice": the configuration is improved by trying neighbours at random and taking the first that improves the score. This approach has several issues:

- The first choice approach is difficult to parallelise efficiently: multiple threads could find improved states, causing a race condition over which updates first. This could waste significant time as threads overwrite each others' improvements, and requires much communication between the threads.
- The algorithm requires a lot of random number generation at a low level. This would be difficult to implement on FPGA, requiring a specialised module to be synthesised.
- The timeout of 1,000 trials without improvement enforces a relatively large lower bound on the amount of work required.
- The algorithm does not avoid repeated work, so it is likely that some neighbours will not be investigated at all, while others will be investigated multiple times. In the case of the ψ wheels, where any configuration has 252 neighbours, of the last 1,000 tests the expected number of times each neighbour is tried is approximately 3.97 (of which 2.97 are useless duplicates), while the probability we miss the best neighbour can be as large as 3.4%[1].

For these reasons, I investigated alternative hill-climbing approaches. One approach, which I will refer to as *wheel-by-wheel*, is shown in Figure 3.1. This prevents work being wasted retrying the same neighbour multiple times, and ensures good coverage of all possible neighbours: it will never terminate in a configuration if a neighbour achieves a higher score. §4.3 evaluates the relative effectiveness of these techniques and others.

For both approaches, the algorithm can be run from multiple random starting points (random restarts) to reduce the chance of finding a local maximum score. This exactly

[1]The probability of missing the optimal neighbour depends on what wheel the optimal neighbour requires changing, since the wheels' varying sizes imply that neighbours are not picked uniformly.

```
Input: wheel_positions, trial_wheels
randomise positions of trial_wheels;
change_made = true;
while change_made do
    change_made = false;
    for wheel in trial_wheels do
        for starting_position in 1..wheel_periods[wheel] do
            wheel_positions[wheel] = starting_position;
            find score of wheel_positions;
        end
        wheel_positions[wheel] = best position found;
        change_made |= (best position != previous position)
    end
end
```

Figure 3.1: Pseudocode for the *wheel-by-wheel* approach to wheel-breaking. Note that useless work can be saved by keeping track of which wheel was last changed, but this is omitted for clarity.

duplicated work is **not** parallelised in this project, although it would be straightforward to do so. This is because this parallelism could always be exploited by running on multiple machines if more performance were required, so it is both more practical and more interesting to investigate the parallelism in the problem at a lower level.

Another algorithmic improvement came from taking the *limitation* into account during χ breaking. In particular, only considering the characters in the *differenced* stream where the χ_2 wheel did not cause the ψ wheels to advance results in the correct χ_2 setting giving a much increased chi-squared score.

3.1.2 Low-level optimisations

This project aims to maximise the efficiency of the codebreaking process. Therefore, it is important to ensure that the code runs as quickly as possible before proceeding to parallelise it. This is where the choice of C as a language pays off, as operation costs are relatively small and predictable (as opposed to Java's garbage collection, for example) and the language allows access to low-level constructs, notably pointers.

Before spending a long time optimising every function, it is important to take into account Amdahl's Law. Amdahl's Law highlights the importance of "making the common case fast", due to the diminishing returns from optimising a rare case. For the codebreaking algorithm, much time will be spent determining the score for different keys. Since we are aiming to optimise for texts of length 6,000, any work that needs to be repeated per-character will therefore dominate the runtime. Figure 3.2 shows the proportion of

Function	Description	Total time spent in function (s)	Calls	Percentage of execution time (%)
`get_wheel_bit`	Extract bit from wheel pattern	2.22	966,140,000	36.9
`advance_wheel`	Rotate wheel one step	1.45	455,181,516	24.1
`get_key`	Extract key bits from machine state	1.29	79,360,000	21.4
`advance_state`	Advance machine state one step	0.61	79,360,000	10.1
`chi_square`	Calculate chi-squared score of text	0.23	15,871	3.8
`apply_crypto`	Coordinate decryption	0.16	15,872	2.7
`set_bit_vector`	Store a bit (representing whether ψ wheels advanced)	0.04	13,820,000	0.7
`get_bit_vector`	Retrieve a bit	0.03	11,282,743	0.4

Figure 3.2: Times spent in different functions in an example run of unoptimised breaking code, as reported by gprof. Only functions representing at least 0.1% of runtime are shown.

time spent in different subroutine calls for an unoptimised version of the code (i.e. a close reproduction of Schüth's Ada implementation). All the functions shown (i.e. all functions representing at least 0.1% of runtime) are either called at least once-per-character or iterate over all the characters. The target of optimisation is therefore very much the decryption and scoring process itself.

The representation of data is a key performance consideration: it should allow efficient processing while being compact and structured to preserve spatial locality, enabling good cache performance. I chose to represent characters of text as `unsigned char`s, despite the 37.5% waste of space resulting from representing five-bit characters with eight bits. This benefits over a more compact representation, e.g. using bitfields, by allowing code to directly access the characters without intermediate bit-shifting operations. Another decision was to represent wheel patterns as 64-bit integers, since fortunately the maximum wheel period is 61, i.e. each wheel pattern is a bit vector at most 61 bits long. This allows a pattern to fit into a single register on a 64-bit machine, which can then be indexed into with a single bit-shift and mask. The inability to access individual bits makes some alternative representations (e.g. an array of `bool`s) overly expensive in terms of cache and register friendliness.

One major optimisation made was to count the frequencies of characters as they are decrypted, rather than decrypting the entire text and then counting the frequencies. This approach uses only half the memory that would be required to store two copies of the text, and the overhead incurred from looping over the text for the second time is

removed. Similarly, merging the `advance_state` and `get_key` functions allows the key to be computed while the wheels are "rotated", which again removes the extra overhead incurred by looping over the wheels twice.

Finally, there was a major performance benefit from pre-computing a buffer of the upcoming wheel settings, such that the next active position of each wheel was represented in the least significant bit of its buffer (Figure 3.3). This allows the key for each input bit to be acquired with a single mask and shift of the pattern buffer, without having to update a position counter for the wheel. When the buffers empty, the new positions of the wheels can be calculated from the known number of times the wheels have rotated, and the buffers refilled.

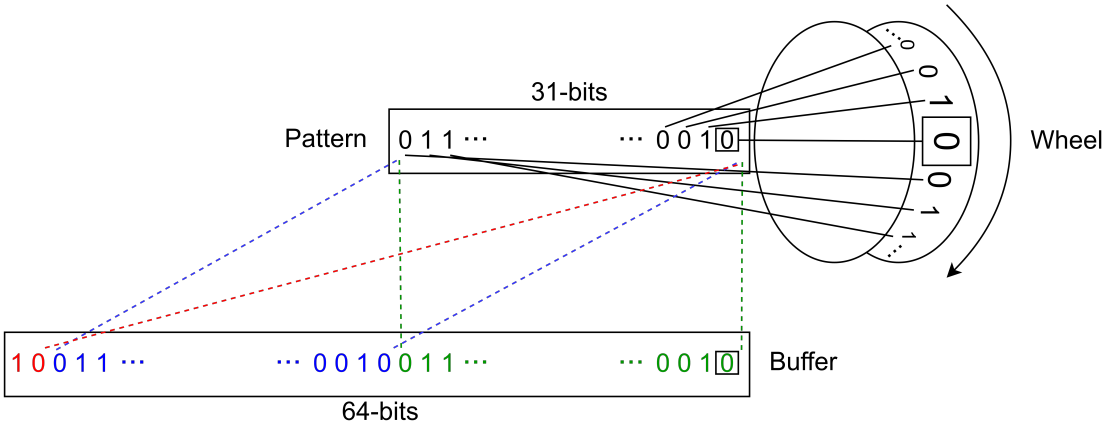

Figure 3.3: The pattern buffer for the χ_2 wheel, assuming it has just been refilled. The boxes mark the active position, i.e. the bit that will be used for the current character. This wheel has period 31, so two full copies of the pattern can be buffered and two bits of one more copy. Different colours show different copies of the wheel pattern in the buffer.

See §4.4.1 for an analysis of how much these optimisations, as well as automatic compiler optimisations, improved performance.

3.2 CPU parallelism

CPUs have traditionally been optimised for single-threaded performance. As transistor scaling has become more difficult and power has become a serious limitation, efforts have instead been made to improve performance by allowing CPUs to exploit parallelism. This has been done through multi-threading – adding several Multiple Instruction Multiple Data (MIMD) cores that can execute independently – as well as vectorisation: adding Single Instruction Multiple Data (SIMD) instructions to operate on multiple independent elements of a "vector". This section discusses the adaptation of the codebreaking algorithm to exploit these features of modern CPUs.

3.2.1 Vectorisation

Modern CPUs typically provide SIMD vector extensions to allow DLP to be expressed concisely by the programmer or compiler. This section looks at the application of the AVX instructions offered by Intel and AMD CPUs to the codebreaking process.

AVX instructions can be used to accelerate code in which the same operation is performed independently on several pieces of data at once. Unlike GPUs, which also exploit SIMD parallelism, AVX instructions operate on a much smaller scale, only parallelising over a small number of values. The programming model provides the user with vector registers of up to 256 bits, which can be operated on in a range of different ways, e.g. as four 64-bit values or thirty-two 8-bit values. Vector instructions express an operation to be performed on each element of these registers. This provides a time and power saving as the operations can potentially be performed concurrently if the functional units are available, and the effort of instruction fetch and decode is amortised over the larger amount of data processing compared to a scalar instruction. However, since support for AVX is architecture-dependent, explicit usage by programmers is uncommon so C does not provide native support for them, instead relying on compilers to convert code to use the instructions (a technique known as auto-vectorisation) where beneficial. This makes manually programming with the instructions somewhat cumbersome, having to use obscure macros – called intrinsics – to refer to the instructions.

Even at the compiler setting for maximum optimisation, the SZ42 code is not automatically vectorised, since there are no loops that can directly be converted to vector versions. Note that the operations on the vector elements must be independent of one another. This rules out perhaps the most obvious method of using vectors – having adjacent input characters as elements – since the key must be calculated sequentially because the current machine state determines whether the ψ wheels rotate, affecting the next key character. Alternatively, elements of vectors could represent the states of decryption using separate keys, allowing several keys to be scored at once. This would require adapting code through multiple function calls, and careful consideration of how to load into and gather statistics from the vectors, essentially doing all of the work done automatically when compiling for GPU for a fraction of the benefit. Instead, I took the approach of expressing the different wheels of a single machine as elements of a vector. This means storing the "pattern buffers" in vectors, using vector operations to perform the appropriate masking to generate the key from their least significant bits, then performing the right-shift to "rotate" the wheels. Figure 3.4 illustrates this process.

To allow use of 256-bit vector registers, I used the AVX-2 extensions. Unfortunately, there are five of each of the χ and ψ wheels: one too many to allow 64-bit pattern buffers to fit into a vector element. Instead, the buffers had to be reduced to 32-bits, increasing the frequency with which they have to be refilled. Nonetheless, vectorisation did improve performance slightly (§4.4.1).

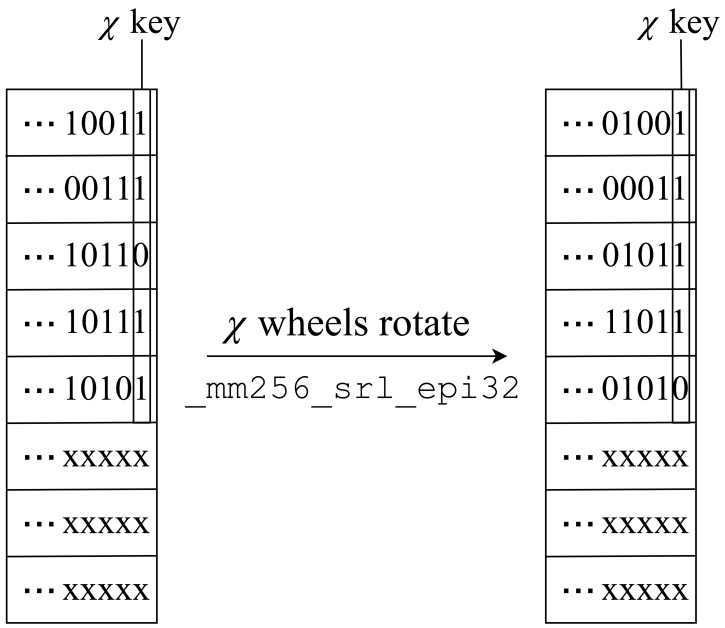

Figure 3.4: A vectorised buffer for the χ wheel patterns. A 256-bit vector is split into eight 32-bit integers, with the first five storing the χ buffers. The remaining elements are unused. This allows the χ wheels to be advanced with a single vector instruction (`_mm256_srl_epi32`: a logical right-shift). Vector operations were also used to aid the process of extracting the least significant bit from each buffer.

3.2.2 Multi-threading

The idea of multi-threading is to split a task into independent "threads" that can operate independently. These can then be interleaved by a processor so that one thread is computing while another is waiting for memory, thus masking memory latency, or even executed simultaneously on a multi-core processor. In principle, the latter implementation can allow an n-times speedup on an n-core machine, although overhead associated with communication between cores can restrict this. Again, Amdahl's Law must be taken into account to determine the overall speedup from the fraction of the workload that is parallelisable.

There are many possible different approaches to parallelising the codebreaking process. Different threads could work on:

1. breaking different ciphertexts;
2. different "random restarts" of the algorithm;
3. scoring different candidate keys;
4. different offsets from start of text;
5. different wheels of an SZ42 machine.

Approaches 1 and 2 are uninteresting and could be achieved by running the process on multiple different machines if more performance is required. Approach 5 would require communication between the threads on every character, reducing the performance improvement. It would also limit the number of threads to twelve (the number of wheels), preventing speedup on machines with a larger number of cores. Approach 4 could be used when working on the χ wheels, where the positions of the wheels can be determined without simulating all of the intermediate steps, since the χ wheels advance every character. However, the approach does not generalise to the ψ wheels, where a large portion of the work of the first thread would have to be simulated to determine the start state of the second thread, etc. Thus, approach 3 was taken. Looking again at Figure 3.2, we see that the vast majority of overall runtime is spent in functions called from the scoring function, so the restriction from Amdahl's Law on the achievable speedup is not a concern.

To parallelise the selected region of the algorithm, a coordinating thread spawns a pool of worker threads, and fills a concurrent-safe circular buffer (based on a guide for a pthreads circular buffer implementation [11]) with job requests. These are expressed as a candidate key, settings to use for the scoring, and a float pointer to which the result can be written (guarded by a mutex and condition variable). Worker threads take jobs from the queue, storing the result to the target pointer when complete and signalling the associated condition variable. The coordinating thread waits on the condition variables until it has more keys to score and there is free space in the job buffer, at which point it processes the scored keys and adds more jobs to the buffer.

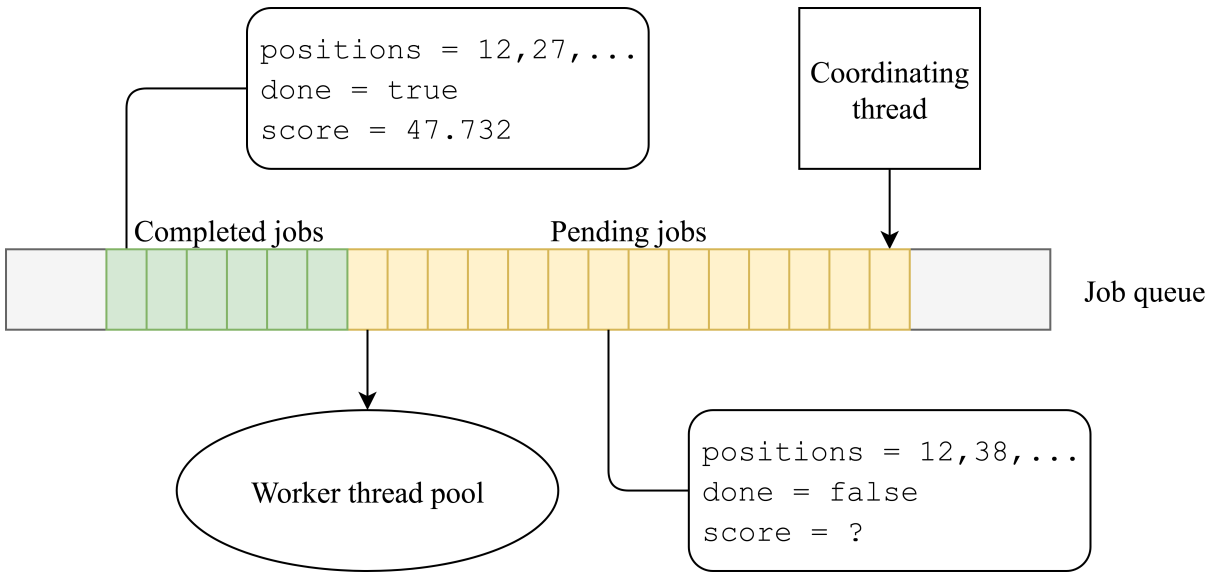

Figure 3.5: Illustration of accesses by threads to a concurrent-safe job queue. Note that job entries contain more fields than those shown to indicate the parameters of the job, e.g. whether to use *differenced* characters.

3.3 GPU parallelism

GPUs offer huge SIMD parallelism, initially designed to optimise the massively parallel workload of computing the colour value of each pixel from a scene description. GPUs can be used when the same operation needs to be performed on a large number of independent elements. This is the case here, since the decryption process is the same for different wheel settings.

Before I could produce the GPU implementation, I had to learn the CUDA programming model and how to write efficient code. I relied on the guides at the NVIDIA website [2] and their programming guide [1]. Hennessy and Patterson's textbook [6] describes how GPUs function, which was useful to tailor code to execute efficiently on the hardware.

3.3.1 Kernel

The model of CUDA is that "kernels" (pieces of code) are started with a specified number of "threads". Each thread runs the same code, but is fed a thread index, allowing it to choose data parametrised on this value. In our case the thread index specifies the starting position of the wheels to be scored.

Kernel code is written in C, meaning the existing SZ42 implementation could be used with only minor modifications. However, to allow GPUs to perform well, it is important to avoid branch divergence, i.e. branches taken by some threads and not others. Branch divergence forces the GPU to execute different threads sequentially rather than concurrently since their instructions differ. Many conditionals can be changed to bit-wise operations, or avoided using other tricks. For instance, consider the following code for advancing a wheel:

```
wheel_position = wheel_position + 1;
//avoid expensive mod operation
if (wheel_position == WHEEL_PERIOD) wheel_position = 0;
```

With a literal compilation, this would lead to huge branch divergence, since different start positions lead to the positions wrapping around at different times. However, since `true` is guaranteed to be represented as 1 and `false` as 0, the code can be rewritten as:

```
wheel_position = wheel_position + 1;
wheel_position -= (wheel_position == WHEEL_PERIOD) * WHEEL_PERIOD;
```

The CUDA compiler can deal with small cases like this automatically, but there were cases where I had to manually change the code to avoid branch divergence. For instance, rather than conditionally calling the `advance_wheel` function for the μ_2 and ψ wheels when they

should advance, I adapted the function to take an additional `enable` argument (1 if the wheel should advance, 0 otherwise), which was added to the wheel position instead of the constant 1. The argument can be determined without conditionals, for instance for the ψ wheels the active μ_2 bit is passed as the `enable` argument.

Another consideration is memory: particularly how the kernels should access the input character stream. Fortunately, CUDA offers "constant memory" for read-only data, which is optimised for the case where all kernels in a thread read the same address at the same time: exactly the case we have here. The read value is broadcast to large numbers of kernels, and the memory is aggressively cached, preventing large memory access latency.

To start a job, the CPU invokes a kernel, specifying the run options (e.g. which wheels to vary, whether to score characters or their differences), and the dimensions of the block. The CPU also allocates the array into which the threads store the result scores. This is copied back to the CPU after the kernel completes for the scores to be processed. It would be possible to reduce the communication between the CPU and GPU by finding the maximum score within the kernel, using parallel reduction [9]. However, profiling shows that the copying of the values and reduction by the CPU represent a small fraction of the total runtime, and so implementing this optimisation would have negligible performance benefits.

3.3.2 Algorithmic changes

Evaluation revealed that the GPU only performs well when very large numbers of keys can be scored concurrently (§4.5). This is because there is some overhead when communicating between the CPU and GPU, and also because the GPU has a lower clock-speed than the CPU and so sees significantly reduced single-threaded performance. The maximum number of keys that can be scored at once when the χ wheels are being broken with a *wheel-by-wheel* approach is 41, much too small to use the GPU to its full potential. Therefore the hill-climbing algorithm has to be changed to increase performance.

The first method I tried is shown in Figure 3.6. This approach has the disadvantage that it no longer guarantees a monotonically increasing score, and so it is possible for the algorithm to get stuck oscillating between two positions. To avoid this, a "sticky wheel" parameter was added, whereby with probability one fifth, a wheel is not rotated to its improved position, meaning that on average one wheel is "sticky" each iteration. With this modification, the algorithm terminates "almost surely", i.e. the probability of it looping forever is zero. Practically, cases where the score oscillates are rare enough that they do not increase the average number of keys scored significantly. With this approach, the number of keys that can be scored concurrently increases to 149 for χ wheels and 252 for ψ wheels: still not enough to utilise the GPU fully. However, this approach unexpectedly led to a smaller probability of finding local maxima, as shown in §4.3.

```
Input: wheel_positions, trial_wheels
randomise positions of trial_wheels;
change_made = true;
while change_made do
    change_made = false;
    for wheel in trial_wheels do
        for starting_position in 1..wheel_periods[wheel] do
            wheel_positions[wheel] = starting_position;
            find score of wheel_positions;
        end
        new_wheel_positions[wheel] = best position found;
        change_made |= (best position != previous position);
        wheel_postions[wheel] = original position;
    end
    wheel_positions = new_wheel_positions;
end
```

Figure 3.6: Pseudocode for the *all-neighbours* approach to codebreaking. Extra parallelism can be exploited as iterations of the outer `for` loop are independent, as well as those of the inner `for` loop.

To further increase the number of concurrent scoring requests made, I changed the approach to a brute-force search on a predetermined subset of the wheels. The χ wheels have relatively small periods, so it is possible to try all the scores for three of them at once. Calculation of the chi-squared score is adjusted to account for there being only eight possible outcomes (i.e. all possible outcomes for the three bits corresponding to the chosen wheels) to prevent interference from the other two wheels. Including the χ_2 wheel in the tested set increases the chance of the configuration with the highest score being the global maximum due to the *limitation*. Otherwise using the two wheels with the smallest periods – χ_4 and χ_5 – keeps the search space size manageable (18,538). Next, all possible settings for the remaining two χ wheels are tested, and at the same time all of the possible settings for the μ wheels. Finally, the ψ wheel settings are found by testing all combinations for ψ_1 and ψ_2 along with all of the combinations for ψ_3 and ψ_4 and all the possibilities for ψ_5, taking the best combinations for these wheels and repeating until no further improvement is found. I refer to this technique as the *GPU-customised* approach: it reduces the GPU invocations to as few as four, scoring 18,538, 3,446, 4,783, and 4,783 keys respectively.

3.4 FPGA parallelism

By designing custom hardware, it becomes possible to exploit parallelism in application-specific ways that may not be possible using general-purpose devices. For instance, wheels can all be updated concurrently while extracting the key, reducing the work of tens of CPU instructions to a single clock cycle on FPGA. Also, modules can be repeated, allowing additional parallelism to be exploited on a higher level. Unfortunately, reconfigurability of FPGAs comes at a price of significantly reduced clock speed: typically hundreds of Megahertz compared to the several Gigahertz for CPUs/ASICs.

For the hardware accelerator, I decided to implement a standalone memory-mapped device. This was chosen over alternative approaches, such as adding an "iterate Lorenz" instruction or coprocessor to an existing processor, since the scoring process takes long enough that the communication will represent a small proportion of runtime. This has the advantage of not relying on a soft-processor design which would prevent use of the HPS and complicate the design process.

The device was subdivided into the following modules, combined as in Figure 3.7:

- `sz42`: Simulates the decryption applied by an SZ42 machine.
- `logger`: Stores character frequencies in the decrypted text to calculate chi-squared score.
- `coordinator`: Provides `sz42` with the key (in the form of wheel patterns rotated to start positions) and ciphertext, and controls parameters of run.

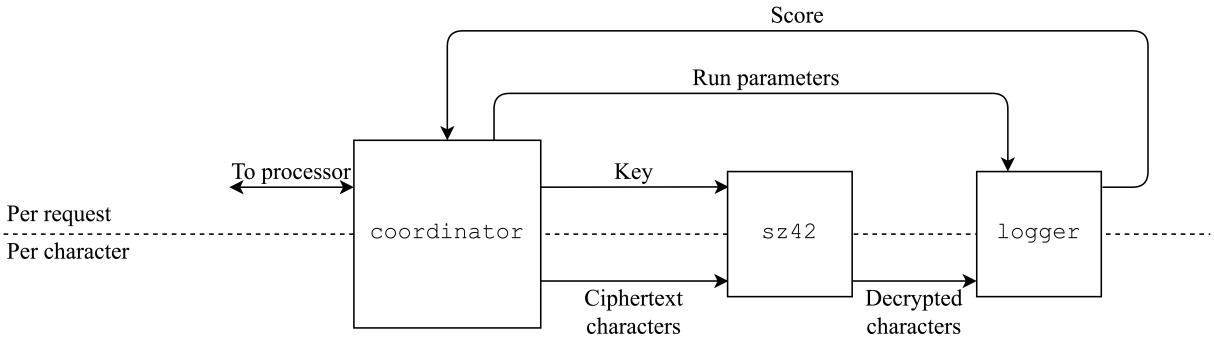

Figure 3.7: Communication between the modules of the codebreaking hardware

3.4.1 `sz42` in SystemVerilog

The purpose of the `sz42` module is to simulate the operation of an SZ42 machine: maintaining state corresponding to SZ42 wheels and computing the XOR of the appropriate

wheel bits with the input ciphertext character. The most naïve approach to doing this would be to store the wheel settings in arrays, and each cycle index into these arrays with a separate wheel-position variable indicating the active index for each wheel. However, this approach would not map well into hardware: a bit-select operation results in the synthesis of a large number of multiplexors. As well as limiting the clock frequency, this would make it more difficult to fit large numbers of `sz42` modules onto a single FPGA, so would limit the potential to extract parallelism.

Instead, I took the approach of storing the pattern for each wheel in a circular shift register. The relevant bit of the wheel pattern is then always at a fixed position in the shift register (index 0), and the next-state logic of each storage cell of the shift register is incredibly simple (Figure 3.8). Note the similarity of this implementation with that of an actual wheel: since the Lorenz SZ42 machine was an actual hardware device (albeit mechanical rather than electrical) it makes sense that its features should be mirrored in our hardware implementation.

This decision has the principal disadvantage of making the logic of loading in wheel-patterns more complex, since the internal representation is further from that used in the CPU. However, when we come to duplicate devices (§3.4.4) the shift register implementation saves area per-device over the indexed array representation, whereas the logic to change the representation is only required once and is shared between all the devices.

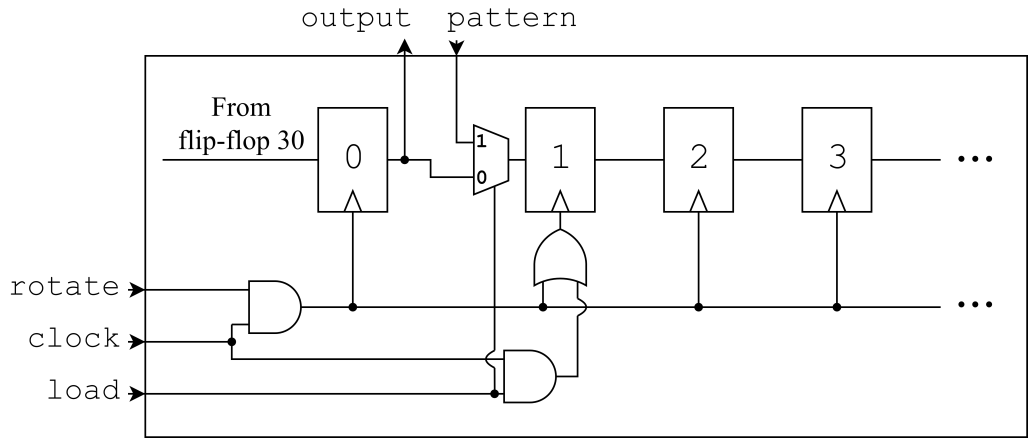

Figure 3.8: The shift register to hold the wheel pattern for the χ_2 wheel (period 31). Each flip-flop takes its new value from the previous flip-flop round the ring if the wheel should rotate: this is indicated by the `rotate` input, which for the χ wheels is always 1. If `rotate` were 0 (e.g. for the ψ wheels), then the flip-flop instead holds its previous value. This behaviour is shown using a gated clock, but is actually implemented using a multiplexor on each flip-flop input since applying logic gates to clocks can cause erroneous clock edges. Only two flip-flop indices are special: index 0 is the the output, i.e. the only value readable from outside the register; index 1 is the input, i.e. `pattern` is loaded into index 1 if the `load` input is set.

3.4.2 `logger` in SystemVerilog

The `logger` module logs the frequencies of characters (or *differenced* characters) so that a chi-squared test can be performed on them as they are decrypted. Not performing this in hardware would require a stream of decrypted characters to be returned to the CPU. This overhead would reduce the benefit of the hardware accelerator. Instead, the aim should be to completely accelerate the scoring process, ideally outputting a chi-squared score and thus minimising the amount of data sent to the CPU.

Unfortunately, the calculation of the chi-squared score requires floating-point arithmetic, which would require additional hardware:

$$\text{chi-squared} = \frac{n\sum_{i=1}^{n} O_i^2}{\sum_{i=1}^{n} O_i} - \sum_{i=1}^{n} O_i \tag{3.1}$$

This is a rearranged form of (2.2) with uniform null hypothesis; $n = 32$ is the number of possible observations and O_i is the number of outcome i observed. One possible approach to avoid the floating-point arithmetic might be to just calculate the frequencies in hardware, and then transfer these back to the CPU to perform the chi-squared test. However, this would still be relatively communication intensive: assuming a maximum of $2^{14} = 16,384$ characters per ciphertext, we would need to transfer $14 \times 32 = 448$ bits back to the CPU per wheel setting checked. While not prohibitive, this starts to add up since we want to include multiple devices on the FPGA to maximise parallelisation. Therefore, I instead decided to determine the sum of the frequencies[2] and the sum of squares of frequencies (14 and 28 bits respectively) in hardware, only leaving the division and subtraction to be done on the CPU.

However, calculating the sum and sum of squares of the decrypted character frequencies directly would require an additional step after decryption. This would be complex to coordinate without limiting the clock frequency. We can avoid these computations by noting that starting with observations O_i, receiving a new observation j yields observations O'_i such that:

$$O'_i = \begin{cases} O_i & i \neq j \\ O_i + 1 & i = j \end{cases} \tag{3.2}$$

$$\sum_{i=1}^{n} O_i'^2 = \left(\sum_{i=1}^{n} O_i^2\right) + 2O_j + 1 \tag{3.3}$$

This allows the sum and sum of squares of the observation frequencies to be calculated while the ciphertext is decrypted using observation frequencies stored on a block RAM on-FPGA. For each character, the relevant frequency can be fetched, used to update the square sum, and then stored with the frequency increased by one. Note the simplicity of this operation in hardware: adding $2O_j + 1$ can be implemented using a single addition

[2]Note that this is distinct from the length of the text for the μ setting where we only count cycles on which the ψ wheels did not advance.

by O_j with a 1 bit concatenated on the end. Implementing this process as additional pipeline stages reduces the clock period overhead of performing the calculations by not extending the critical path.

3.4.3 `coordinator` in SystemVerilog

The `coordinator` module bridges the CPU and the other two modules, converting the key between the different representations used by the two devices. Specifically, it provides to the CPU an Avalon Memory-Mapped interface, allowing the CPU to write to specific "memory addresses" the patterns to be loaded into each wheel. The CPU signals that the process should start by writing to a particular control register, while also specifying the configurations to be used for the run (e.g. whether to use *differenced* or non-*differenced* characters). The coordinator then "rotates" the wheel settings and sends them to the `sz42` module to be loaded into the shift registers one "slice" at a time.

When the scoring process is complete, the `coordinator` signals this to the CPU by setting a status register polled by the CPU. The CPU can then access the results (in the form of the sum of frequencies and sum of squares of frequencies) via memory-mapped reads. In software, writing a short function to issue a given request and block until the result has been computed and loaded back from the FPGA allows the C code from the baseline implementation to be reused with minor modifications.

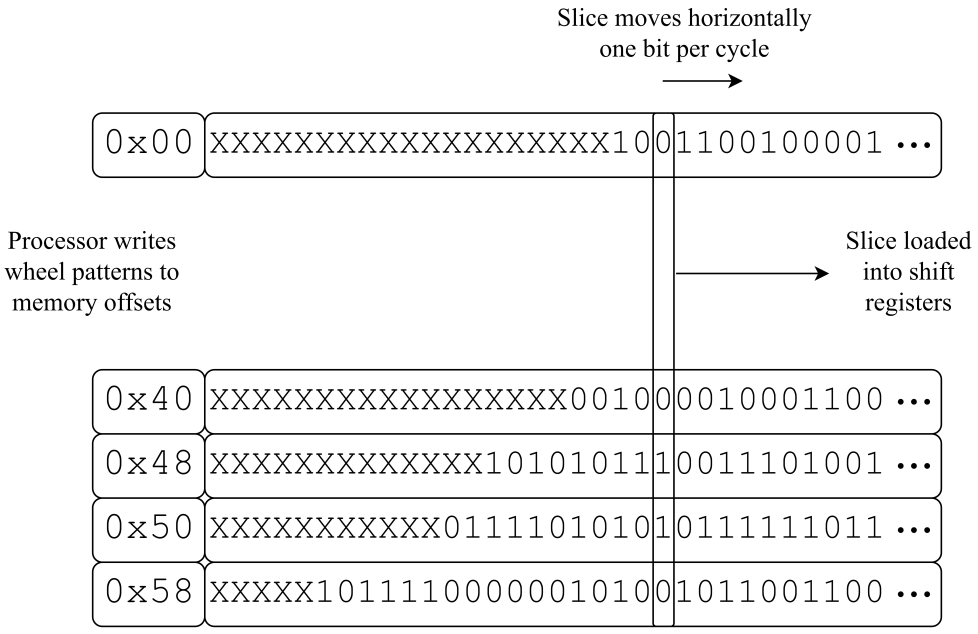

Figure 3.9: The load phase of the coordinator. Patterns written vertically by the processor are fed horizontally into the `sz42` shift registers. Note that earlier undefined values (past the period of the wheels) are overwritten by the later parts of the patterns. Address `0x60` is used as a status register the CPU can access to start a run and specify the run parameters.

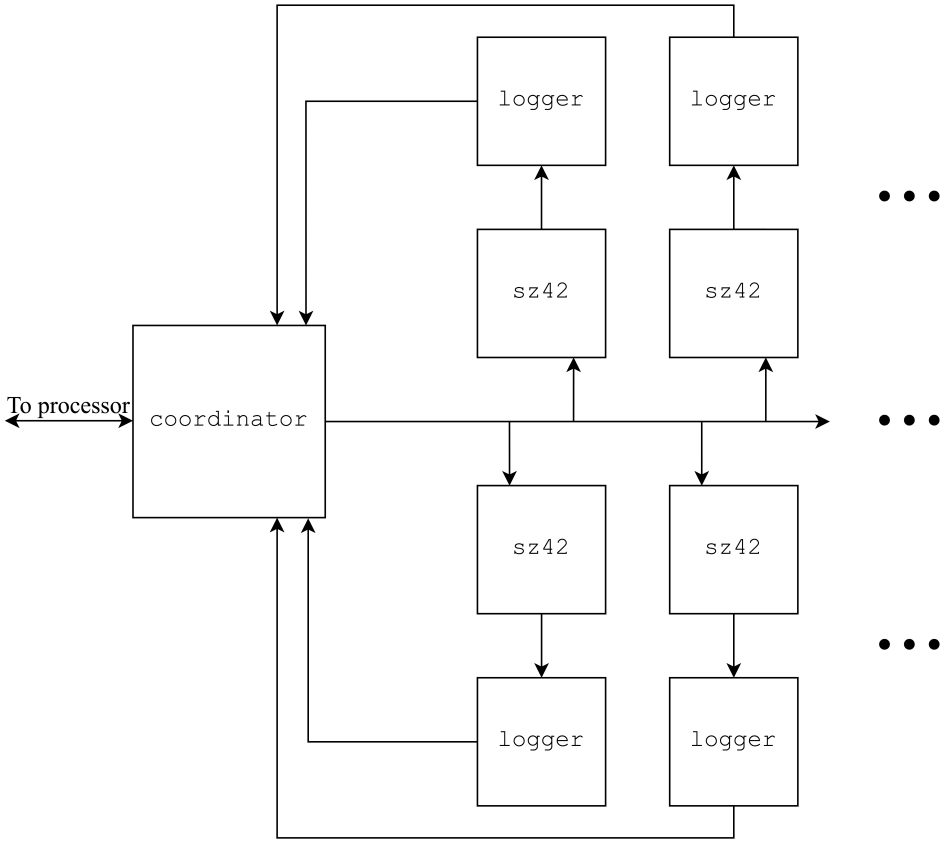

Figure 3.10: One coordinator shared between multiple codebreaking modules. Note that the inputs to all the modules are identical; differences in their internal shift registers make their output differ. The signals are the same as in Figure 3.7, but the details have been omitted for clarity.

3.4.4 Duplicating devices

A key advantage of custom hardware is that it can be very compact, potentially allowing devices to be duplicated to exploit higher-level parallelism. In our case, 61 (the maximum period of any wheel) `sz42` and `logger` modules can be synthesised onto a single FPGA. This allows the scores for all possible positions of a particular wheel to be calculated concurrently. However, this required an extra pipeline stage to be added to allow the ciphertext character to propagate from the coordinator to all of the `sz42` modules, since otherwise this becomes a critical path.

Duplicating the `sz42` modules is pointless if they all perform exactly the same operation on the same data. Therefore, some modification to the design is required to differentiate the work performed by each module. I chose to have each device calculate the score for a different position of a given wheel, since this operation is required in both the brute-force search and the *wheel-by-wheel* search techniques. To achieve this, I modified each `sz42` module's shift registers to each have two outputs: one from index 0 (as before) and one from an alternative index determined as an elaboration-time module parameter (Figure 3.11). The selection between these two outputs is performed based on

a signal indicating whether this wheel is under test: if so then the alternative index is used, otherwise index 0. The modules are synthesised in a `generate` loop, each having a distinct alternative index parameter between 0 and 60, thus allowing all positions of one wheel to be scored at once. The only changes to the CPU code required are that it must indicate which wheel is under test and that it must read in an array of return values (rather than just one) when scoring is complete.

Another possibility for exploiting additional concurrency would be to duplicate these devices, since the FPGA area used for the cryptographic modules themselves is relatively small. This would allow multiple configurations to have all positions for one wheel scored, all concurrently. However, I did not implement this due to the coordination and software complexity it would incur.

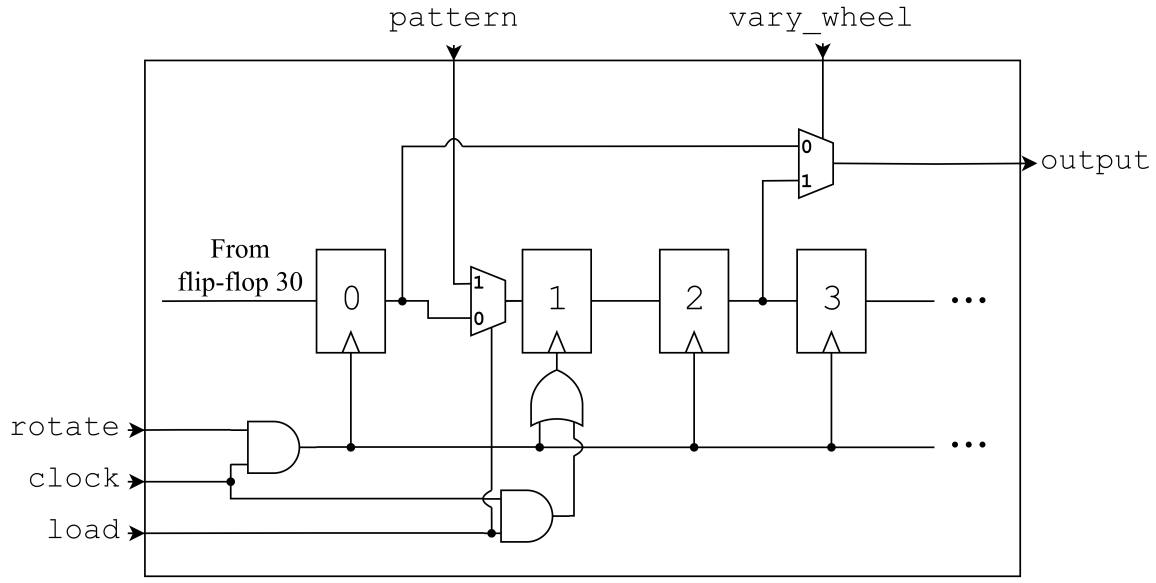

Figure 3.11: Modified version of shift register from Figure 3.8. If the scoring request is for the best position of the χ_2 wheel represented by this shift register then the `vary_wheel` input is set. This causes this shift register to draw from index 2 instead of index 0, so the wheel has effectively been shifted forwards two positions. Other duplicated `sz42` modules have the alternative output drawn from a different index of the shift register, and so score different starting positions of the wheel.

Chapter 4

Evaluation

This chapter examines the performance of the different approaches. I first directly compare the overall performance of the different implementations against the Ada and single-threaded C baselines (§4.1). Next, I compare the performance at a lower level via the throughput of individual score calculations (§4.2). §4.3 discusses the effectiveness of the different hill-climbing approaches considered. Finally, I examine the key performance considerations of each implementation individually (§§4.4-4.6).

Appendix A details the methodology behind each graph. Note that all error bars show a 95% confidence interval for the mean, i.e. 1.96 standard deviations for times (although these are often too small to discern due to the large number of repeats performed) and the corresponding beta distribution value for probabilities for which the normal distribution approximation can break down when close to 0 or 1. Appendix C details the plaintexts used. Note that the competition texts are treated as an entire population for the sake of statistics, whereas the other texts are treated as samples.

4.1 Overall effectiveness on different texts

The overall effectiveness of each implementation can be measured as the speed and reliability on different text types. The aim set in the requirements analysis was to be at least as reliable as the Ada implementation [10], and otherwise be as fast as possible.

The runtime requires more explanation: the model assumed is that we have a large number of ciphertexts, aiming to break as many as possible per second. We therefore measure the time spent actually computing on each text, but not the time taken for initialisation. For example, the time taken to initialise the GPU and load the ciphertext into its memory is not included in timing, since the GPU could potentially be left initialised between multiple codebreaking runs, with new texts loaded in parallel with computation on previous texts. The loading of texts into the FPGA RAM is not included in the timing

for the same reason. However, the communication latencies between the GPU/FPGA and the coordinating CPU are included in the timing as these are an integral part of the codebreaking process.

The reliability refers to the probability of the found key being correct. As noted in the preparation, it is impossible to guarantee this is 1 for all texts. Also, for a given text, different seeds for the random number generator can give rise to different results, since different starting positions are used and different neighbours explored (except for the GPU implementation, which is deterministic). Therefore, the success of the process depends on the text being scored and the seed used. The success probability on each text is the average of a large number of tests (between 10 and 100) on 100 randomly generated texts of that type, except in the case of the competition texts of which only three are available. Confidence intervals are constructed from the data, taking into account the number of repeats performed.

It is clear from Figure 4.1 that all of the implementations are more reliable than the Ada baseline implementation: this is mainly due to the algorithmic improvement of taking the *limitation* into account during χ breaking. Note that the single-threaded C, multi-threaded C, and FPGA implementations are all algorithmically identical: using the *wheel-by-wheel* approach for the χ and ψ wheels, and a brute-force search for the μ wheels. In the case of the two C implementations, the random number generation mechanisms (C++'s Mersenne Twister implementation) and seeds used for the starting positions of the wheels are also identical, hence the success probabilities of the two programs were guaranteed to be the same. Due to limitations of the HPS, the FPGA output used a different random number generator (C's `rand()`), but the graph shows that the probability of successful decryption was unaffected. Note also that the errors in the success probability are correlated, since most of the uncertainty arises from the distribution of the texts and the same texts were used to test each implementation. This explains the 95% confidence intervals being so much larger than the observed variation between the three implementations. The GPU implementation is significantly more reliable, especially on the harder random words text, due to the deterministic *GPU-customised* approach taken. §4.3 examines the differences between the searching approaches in more detail.

All implementations were faster than the Ada implementation, due to some combination of algorithmic improvements (§4.3), low-level improvements (§4.4.1), and increased parallelism. As expected, the multi-threaded version is faster, although not by a full factor of four: §4.4.2 gives more detailed analysis. The GPU version achieves further speedup through massive DLP. The FPGA achieves even greater speedup by extracting parallelism at a lower level, as well as extraction of DLP.

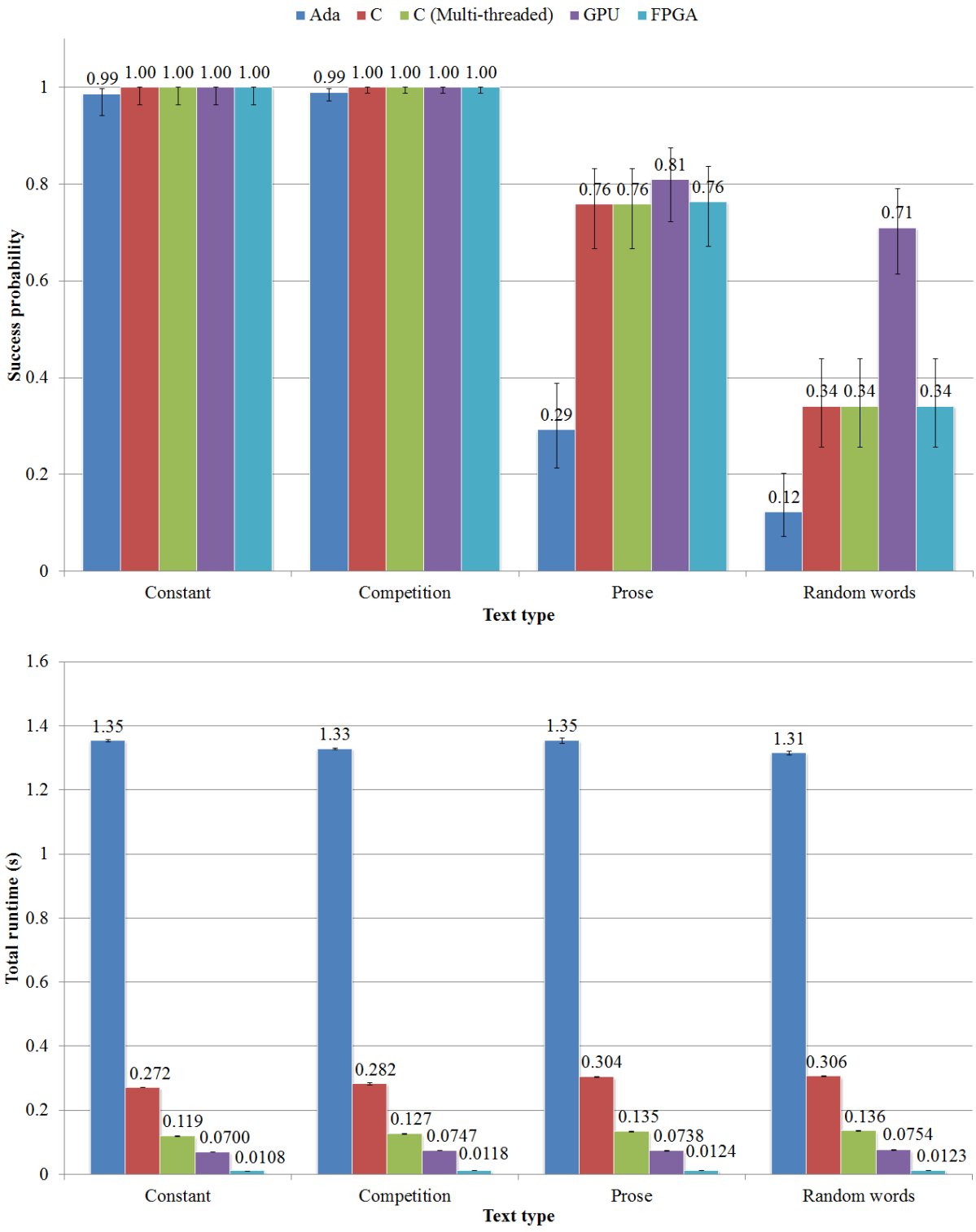

Figure 4.1: The overall speed and reliability of the different implementations and the Ada baseline on four different types of text. Times shown are the mean times taken to completely decrypt a single text, and the success probability is the fraction of keys guessed correctly. Appendix A details the hardware and compilers used; note the CPU has four cores and the FPGA clock frequency is 133.33 MHz.

4.2 Comparison between devices

In addition to the overall effectiveness of the implementations, it is interesting to measure the maximum performance of each device in terms of the maximum scoring rate (throughput). This is a maximum because, for example, we assume keys can be scored in large batches (relevant for the multi-threaded and GPU implementations). We also assume the scores require varying the highest-period wheel, since otherwise some of the scores generated by the FPGA are wasted.

Figure 4.2 shows the measured throughputs of the different approaches, reporting the rate at which the scores can be returned to the coordinating device/thread to be used. For instance, this includes the latency associated with communicating from the CPU to GPU, HPS to FPGA, etc.

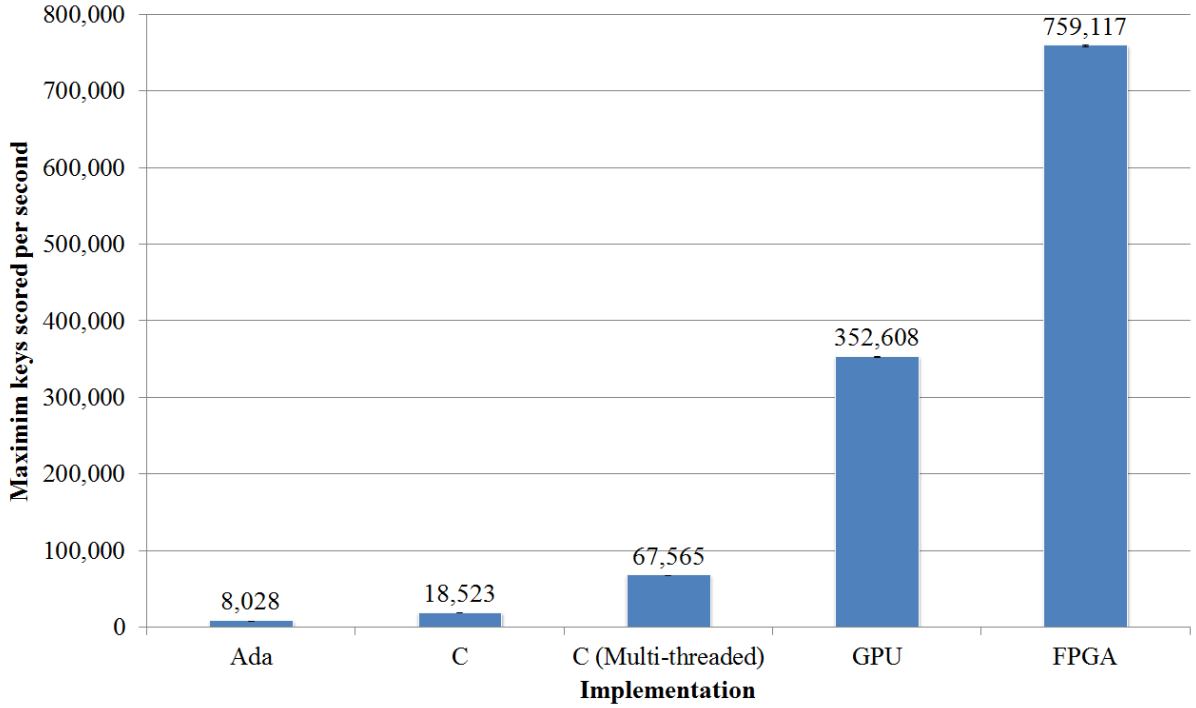

Figure 4.2: Maximum throughput of each implementation. Again, the CPU had four cores and the FPGA was synthesised at 133.33 MHz.

The four-way multi-threaded C implementation achieves a 3.6-times speedup over the single-threaded implementation, close to the maximum factor of four. However, as discussed in §4.4.2, this does not translate into a 3.6-times improvement in the overall codebreaking process. The GPU implementation achieves a significant speedup over the CPU implementations via DLP, but this is only achieved when large batches of keys can be issued concurrently (§4.5). This restricts the high-level algorithms that can be used, preventing the speedup from translating directly into an overall performance benefit. The FPGA provides even larger speedup without such restriction.

4.3 Comparison between hill-climbing approaches

The Preparation and Implementation chapters discussed several different approaches to searching the possible settings for the χ and ψ wheels:

- Brute-force search: score every possible setting and take the maximum.
- Monte Carlo search (Schüth): change a random wheel to a random position and check for an improvement.
- *Wheel-by-wheel* search: choose the best position for each wheel in turn.
- *All-neighbours* search: find the best position for each wheel with the others unchanged, then update them all simultaneously.
- *GPU-customised* search: brute-force searches on predetermined subsets of the wheels.

The different approaches can be evaluated using several different metrics. The total work done can be measured as the total number of keys that were scored. The critical path length (span) can be measured as the maximum path of keys that had to be scored sequentially. This gives an idea for the parallelisability of an algorithm as it provides a lower bound on the total time taken to determine the key. Finally, the success probability measures the probability that a run finds the correct key. There are three ways the algorithms can fail to find the correct setting:

- Bad scoring: the correct key may not give the global maximum chi-squared score. This case cannot be fixed without fundamentally changing how keys are scored, e.g. by assuming a particular plaintext distribution.
- Local maxima: the algorithm can find a key that attains a local maximum score, i.e. all of the "neighbouring" settings (settings differing in the position of exactly one wheel) lead to a reduced chi-squared score.
- Missed neighbour: the algorithm may miss a neighbour setting that represents an improvement, instead terminating on a non-maximal setting.

Figure 4.4 shows the average success probability, total runs, and critical path lengths for the different techniques on four different types of text, each having different "difficulty" arising from its statistical properties. Appendix C details each type of text. Note the brute-force method does not give 100% success probability due to "bad scoring" cases where the correct setting does not give the maximum chi-squared score.

The diagram justifies the choice of a *wheel-by-wheel* approach for χ-breaking on CPU. In all cases, the *wheel-by-wheel* success probability is within 20% of that for Monte Carlo,

Approach	Total work	Parallelisability	Success probability
Brute-force	Size of entire space.	Embarrassing: all settings could potentially be scored in parallel.	Very good, only fails in case of bad scoring.
Monte Carlo	At least 1,000 due to Timeout.	Poor: if parallelised, only improvement from one thread can be used.	Can fail due to all three reasons.
Wheel-by-wheel	At least sum of periods.	All settings for a wheel can be checked concurrently.	Cannot fail due to missed neighbours. Local maxima more likely.
All-neighbours	At least twice sum of periods, often much more.	Sum of all wheel periods can be run concurrently.	Can fail due to all three reasons (neighbours can be missed due to random "sticky wheel"). Local maxima less likely.
GPU-customised	At least 31,550 (for all wheels).	Excellent: critical path as short as four.	Deterministic, so only fails to bad scoring (although these are slightly more frequent due to different scoring metric).

Figure 4.3: Qualitative comparison between hill-climbing approaches

but scores less than a third as many keys. Hence, the *wheel-by-wheel* setting can be performed from three random starting positions for each Monte Carlo search, giving a much higher success probability for less overall work.

Note the increased success probability the *all-neighbours* approach achieves for the more difficult texts over the other probabilistic approaches. This is because there is some probability for each wheel that, from a random setting, the highest-scoring position is not the correct one. In alternative approaches where this new position is taken immediately, this may prevent finding the correct position for other wheels, as they do not correlate with the incorrect position of the first adjusted wheel. By postponing the update until the best position for all wheels has been found, it is more likely that at least one of them is correct, at which point the others will score more highly when their correct position is tested. However, the "sticky wheel" adjustment forces this approach to have a maximum success probability of about 90%, even on the easiest of texts.

The *GPU-customised* approach achieves significantly greater success probability due to the sheer number of keys scored. However, the search is deterministic, so random restarts are not possible. The short critical path length is the main motivation for this approach.

The μ and ψ settings are much easier to break than the χ settings, with a brute-force search and *wheel-by-wheel* search successfully finding the correct μ and ψ setting respectively in every case tested above. For the ψ setting, the non-*differenced* characters

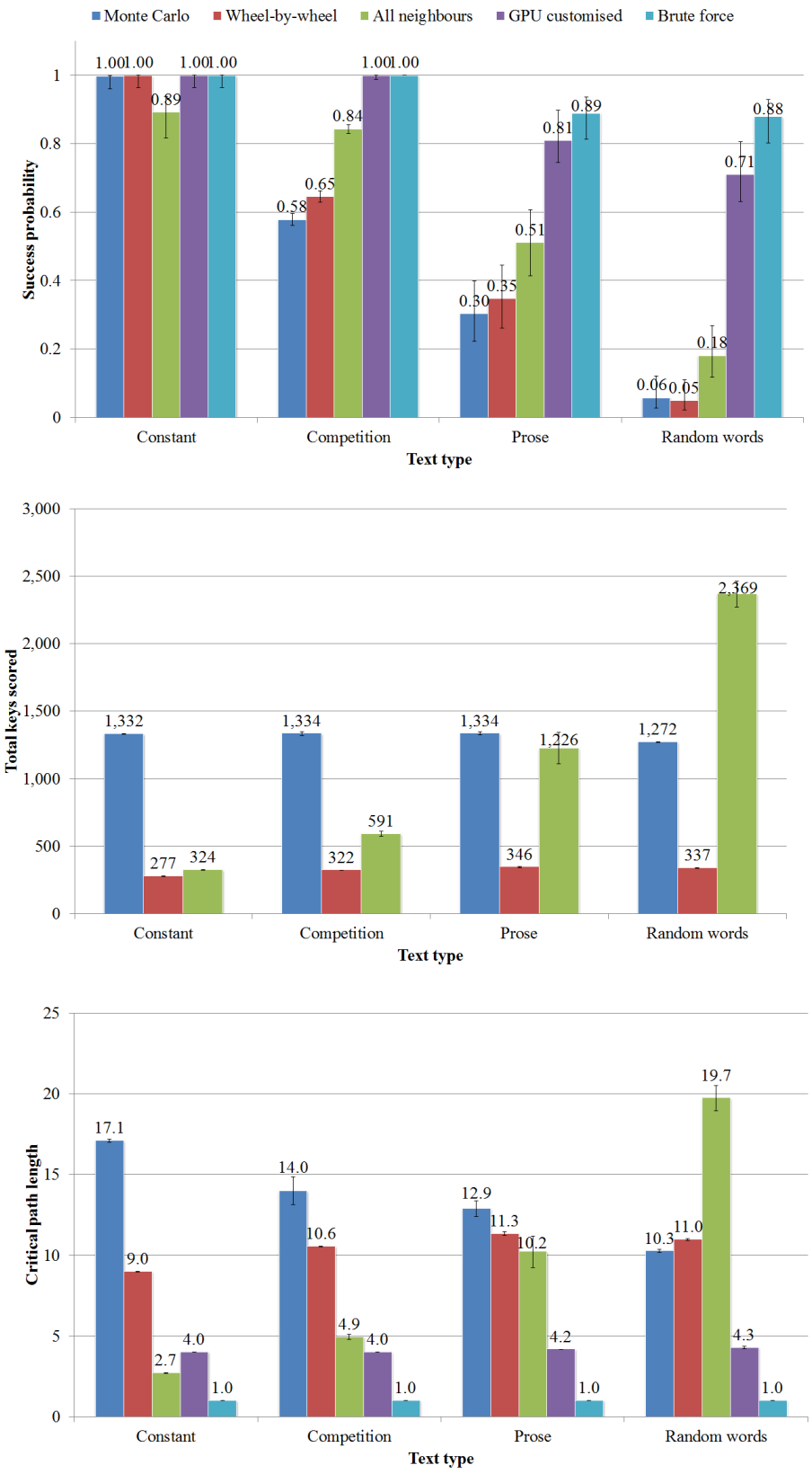

Figure 4.4: Effectiveness of different hill-climbing algorithms at finding χ settings for different texts. Note the brute-force total keys scored (22,041,682) and *GPU-customised* total keys scored (between 31,550 and 36,333) have been omitted so other values are visible.

in all the texts used have such strong non-uniformity that the correct settings can be found for each individual bit independently.

4.4 CPU

4.4.1 Low-level optimisation

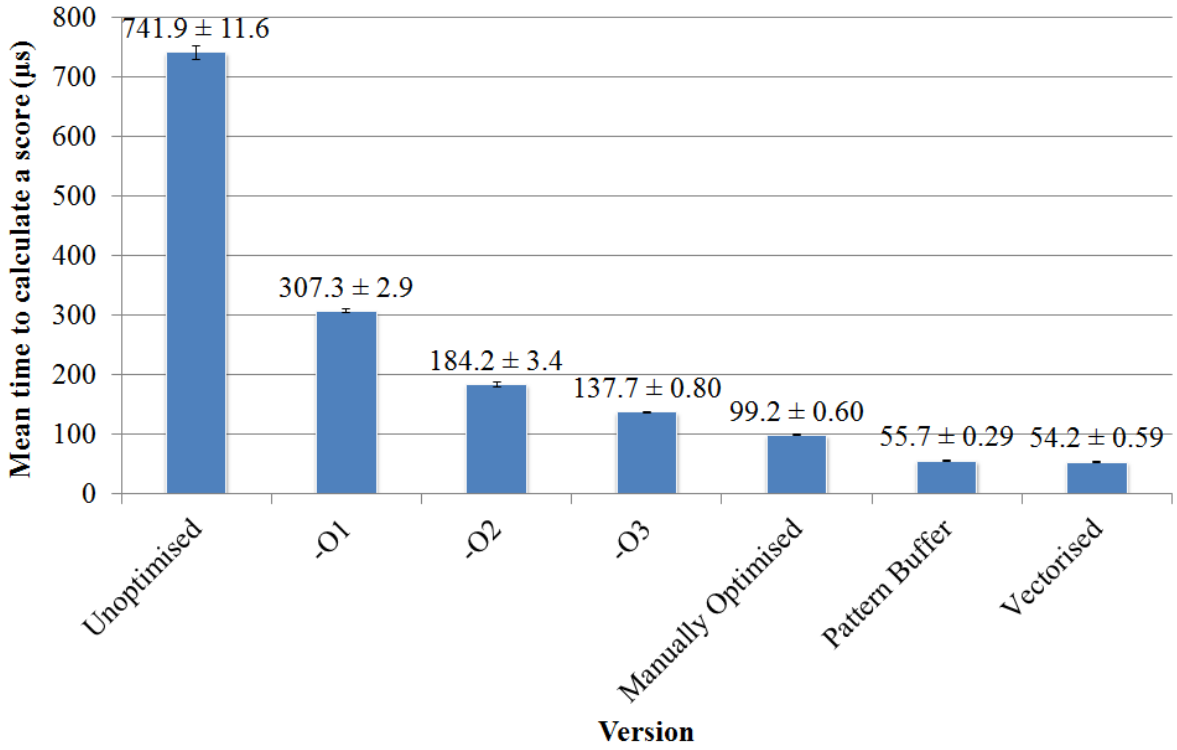

Figure 4.5: Cumulative performance improvements to key-scoring process from successive levels of optimisation, both automatic and manual.

§3.1.2 discusses the particular optimisations, and §3.2.1 details vectorisation.

Figure 4.5 shows the power of compiler optimisation, but manual reworking of the code was required for the best performance. The "Manually Optimised" version refers to that with the multiple loops over the text removed and merged `get_key` and `advance_state` functions. The "Pattern Buffer" version was only changed by adding the preloaded buffer of upcoming wheel bits, reducing the amount of bookkeeping necessary to keep track of wheel positions. The vectorised version sees a slight performance increase, despite a very limited portion of the code being vectorised, and the reduced buffer length necessitated by small vector size.

4.4.2 Multi-threading

The effectiveness of the multi-threaded implementation can be measured via the effect of the number of cores on the breaking speed. I measured the speed of the overall breaking process, since it is the coordination between the threads that is most important when determining the scalability of the implementation.

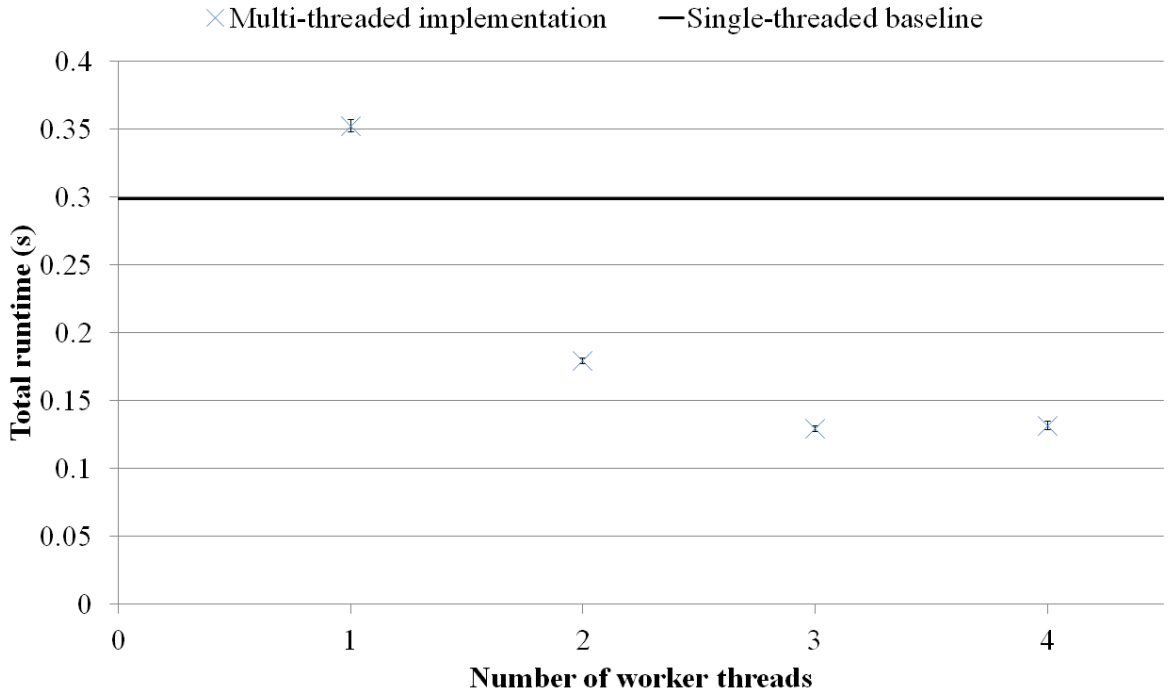

Figure 4.6: Overall speed of the multi-threaded implementation with different numbers of worker threads. The time taken by the implementation without any multi-threaded primitives is shown for comparison. This was carried out on a four-core CPU.

Figure 4.6 shows the relationship between the number of worker threads and time taken to completely break a cipher. The point with one worker thread gives insight into the constant overheads associated with communicating between a coordinating thread and worker thread and maintaining a queue structure. The points for two and three worker threads show just below a two- and three-times speedup respectively, as expected. The fourth worker thread does not improve performance, likely due to contention with the coordinating thread and other background processes running on the system.

The implementation will not achieve an n-times speedup with n worker threads for large n, even on a machine with more than n cores. The main reason for this is work-balancing: for large parts of the process (e.g. χ breaking using a *wheel-by-wheel* approach) only a small number of concurrent key-scoring requests are made based on the period of the wheel (between 23 and 41). If n does not exactly divide this number then some threads will waste time waiting for others to complete. Furthermore, the concurrent-safe queue

is a single point of contention between the threads, which will increasingly bottleneck performance as n becomes very large. Therefore, the high-level algorithm would need significant rethinking – becoming more like the GPU approach – to exploit very large numbers of cores.

4.5 GPU

The key performance consideration with GPUs is the requirement to dispatch a large number of jobs, enabling parallelism to compensate for each individual job being slower than on a CPU implementation. Figure 4.7 demonstrates this: the number of keys scored concurrently in the *wheel-by-wheel* method used in the CPU is at most 61, requiring orders of magnitude more time per key than can potentially be achieved. In fact, as seen in Figure 4.8, batches smaller than 1,500 keys lead to the time spent per key being greater than 4 µs, whereas the graph shows that with larger batches, times of below 2 µs per key can be achieved. This justifies the *GPU-customised* method, which issues batches of sizes between 3,446 and 18,538.

Figure 4.9 shows a timeline of overall codebreaking execution using the *GPU-customised* approach.

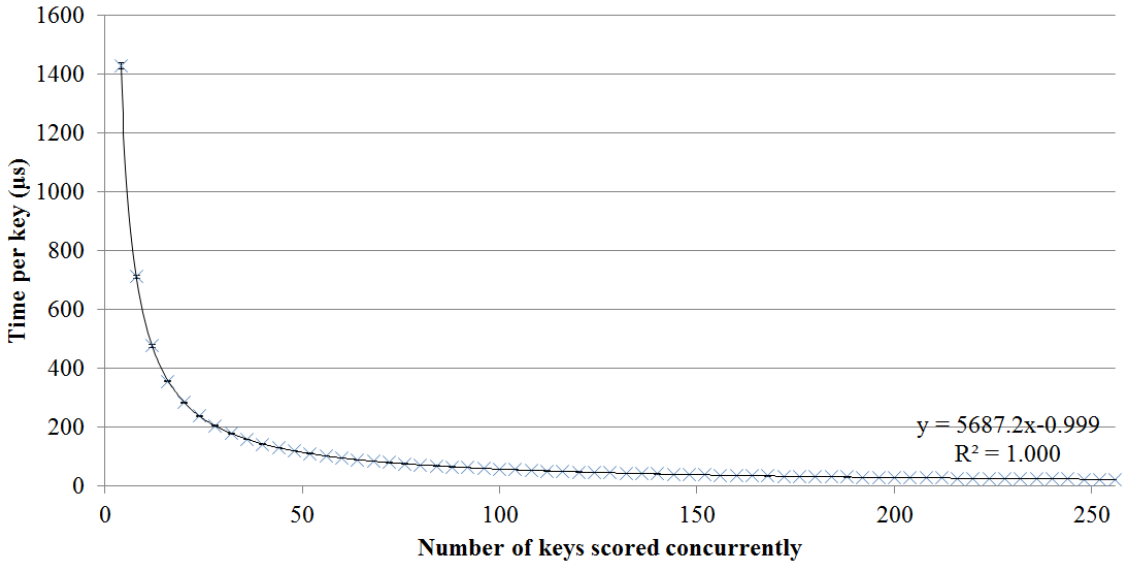

Figure 4.7: Time taken by the GPU per key scored for small batches of keys. The trendline shows a power-law fit with very strong correlation, showing that the time per key is inversely proportional to the size of the batch. This is because the time taken per batch is constant (5.69 ms) for batch sizes this small, so this time is amortised over more keys as the batch size increases.

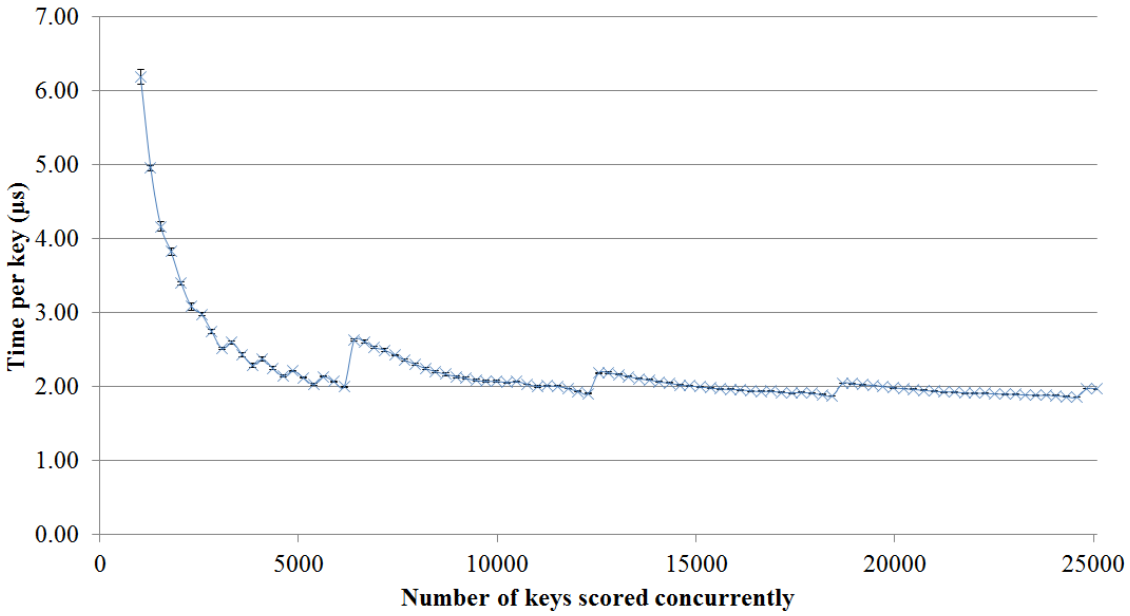

Figure 4.8: Time taken by the GPU per key scored for large batches of keys. Note the large jumps as the GPU is forced to execute an additional group sequentially, followed by a gradual decline as this additional group's execution time is amortised over more keys.

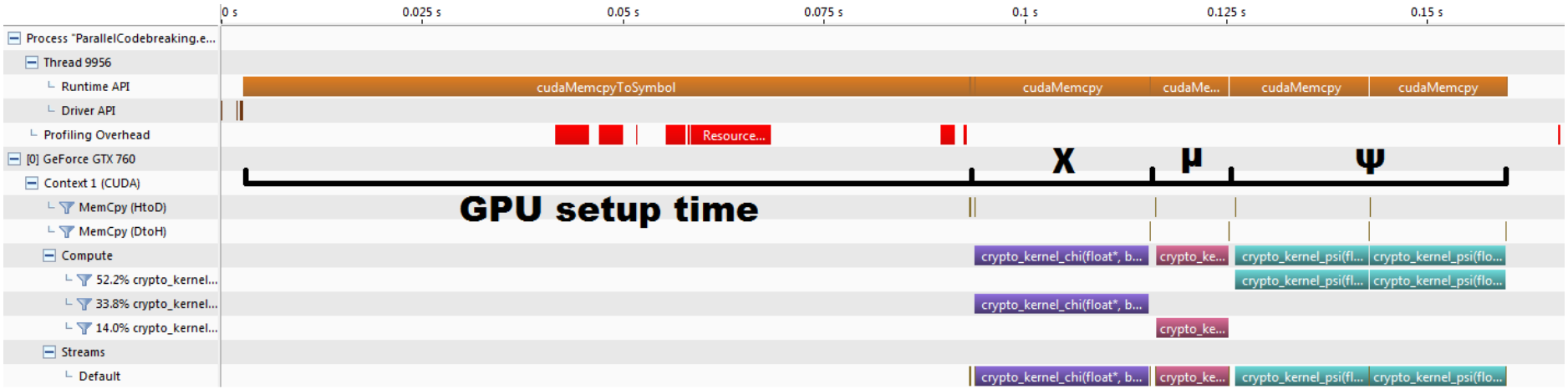

Figure 4.9: Timeline of GPU codebreaking process, as reported by CUDA's Visual Profiling Tool. Note the initialisation operations (implicitly counted in `cudaMemcpyToSymbol`) take up more than half of the total runtime. Otherwise, the GPU achieves high utilisation with communication overhead taking up a small fraction of runtime.

4.6 FPGA

This section examines the performance of the FPGA hardware accelerator synthesised on a DE1-SoC Cyclone V board.

Figure 4.10 shows Quartus' reported synthesis statistics. Note that the design compiled includes the additional modules required to interface with the HPS, and a Phase-Locked Loop (PLL) to allow higher frequencies to be generated from the 50MHz input clock.

Maximum frequency	143.39 MHz
Logic utilization (Adaptive Logic Modules used)	13,339/32,070 (42%)
Total registers	39,653
Total block memory bits	110,436/4,065,280 (3%)
Total RAM blocks	77/397 (19%)

Figure 4.10: Statistics reported by Quartus for the synthesised FPGA design. The synthesis was carried out using the "aggressive performance optimisation" compilation setting.

Each scoring request can theoretically be completed in 6,140 cycles: 75 to load the patterns into the coordinator and then into the shift registers; 6,000 to process a 6,000 character text (plus 3 to clear the pipeline); and then 62 to return the result (assuming the wheel under test has period 61). The maximum frequency of 143.39 MHz therefore allows the device to answer a scoring request on a 6,000 character text in 42.8 µs.

However, the FPGA implementation does not perform this well in practice because the FPGA must wait for requests from the CPU before starting, and so does not spend all of its time computing. This can be quantified: taking the example of the prose text scoring, the average number of requests made to the FPGA per cipher broken was 169. Therefore, the total time spent working by the FPGA was $\frac{169 \times 6140}{133.33\,\text{MHz}} = 7.78\,\text{ms}$ (since the version evaluated was synthesised at 133.33 MHz). Of the total time of 12.4 ms, this therefore represents an FPGA utilisation of only 63%. The remaining time is spent processing by the HPS, or waiting for communication between the HPS and FPGA.

Figure 4.11 shows the placement of the design. Note that mostly each `logger` module is synthesised as a single block due to the close interdependence between the different components. However, the `sz42` modules themselves are split up by the compilation process due to the independence between the separate character bits in the encryption process.

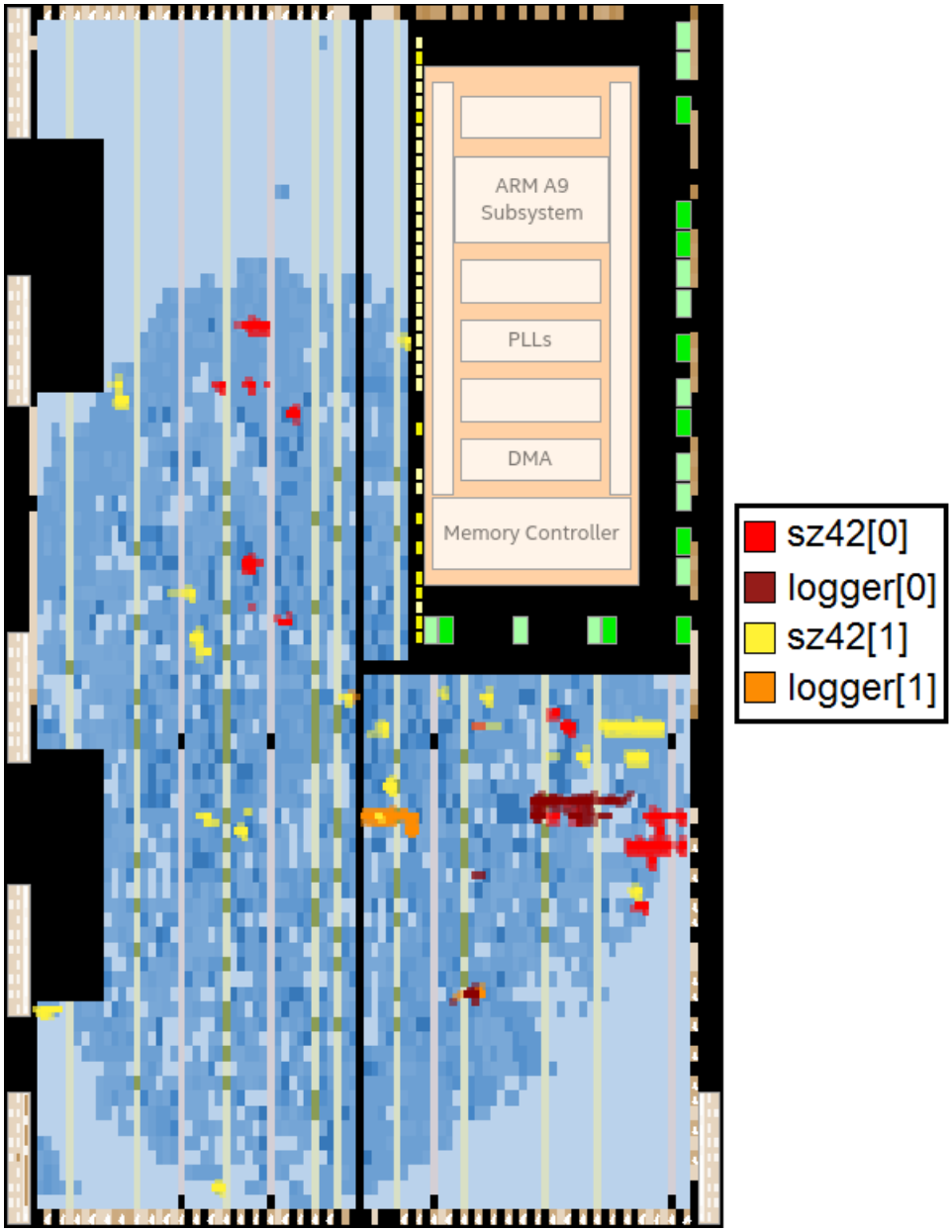

Figure 4.11: Output of Quartus' "Chip Planner" for the synthesised design. Two particular `sz42` modules and corresponding `logger` modules have been highlighted. Otherwise, darker shades of blue correspond to greater usage of the FPGA.

Chapter 5

Conclusion

Codebreaking of the Lorenz cipher used during the Second World War has been investigated, starting with a prior algorithm written in Ada [10]. The algorithm was reimplemented in C: a combination of algorithmic improvements (§3.1.1) and low-level optimisations (§3.1.2) led to a four-times speedup over the Ada version, as well as a significant reliability increase on harder texts. A multi-threaded C version (§3.2.2) and a GPU version were written (§3.3), giving a further two-times and four-times speedup respectively. A vectorised CPU version was also produced, giving modest speedup (§3.2.1). Finally, a hardware design was produced on FPGA (§3.4), providing a 25-times speedup compared to the single-threaded C code. The FPGA version was an extension of the project that required a great deal of effort to design and verify: this was a large fraction of the work completed in this project. Thorough evaluation was performed, including a comparison between devices (§§4.1-4.2), an investigation of the different algorithmic approaches (§4.3), and the device-specific considerations for each implementation individually (§§4.4-4.6). Thus, all the goals from the proposal were fully met.

5.1 Discussion

One perhaps surprising aspect of the project is the extent to which the codebreaking algorithm had to be adapted (e.g. in the different hill-climbing approaches) to extract enough parallelism to occupy the full capacity of the GPU and FPGA. This highlights the difficulty in providing auto-parallelisation and high-level synthesis tools: often the high-level algorithm needs to be changed to support the low-level hardware in ways that are practically impossible to discover and verify automatically.

One outcome of the evaluation is an answer to the thought experiment of how Bletchley Park would have broken Lorenz were it operating today. To some extent this comparison is unfair: the Lorenz machines were designed and built subject to computational restrictions of the time. However, given the importance of codebreaking to the war ef-

fort – as seen in the development of novel computing machines such as Colossus – it is reasonable to suggest that it would have been worth investing in ASICs. This means the custom hardware would be even further ahead due to the scaling up in performance from an increased clock speed (although it is worth asking just how many ciphertexts would need to be broken, given the speed of all of these methods).

A contemporary analogy to this problem can be seen in Bitcoin mining: initially mining could be performed on a CPU, but then miners had to move to GPUs as mining became more difficult. Custom ASICs have now practically become a necessity to make money mining Bitcoin, due to the power efficiency of bespoke hardware and the speed arising from large parallelism. The process of mining Bitcoin – evaluating a cryptographic function on a large input then testing a property of the result – is not too dissimilar from the application of breaking Lorenz ciphers discussed in this project.

Another interesting angle is a comparison with how Colossus itself operated. Colossus was an electronic device, able to apply a key to a ciphertext and then compute some programmed boolean predicate on each character and count the number of times it was true [4]. The speed-limiting factor was the paper input tapes, which allowed 2,000 characters to be decrypted per second. Rather than computing an arbitrary boolean predicate (which would allow the plaintext distribution to be taken into account), all of my implementations use a chi-squared test, like Schüth's. The FPGA implementation is most similar to the Colossus implementation: both compute the scores for all positions of a given wheel, holding the key in shift registers. The FPGA allows the text to be stored in BRAM, allowing 133,300,000 characters (one per cycle at 133.33 MHz) to be fed into the cryptography modules per second. This combined with the additional parallelism – scoring all the positions for a wheel concurrently instead of just five at a time – gives a theoretical speedup factor of the FPGA implementation over Colossus of roughly 813,000. In addition, coordination of different Colossus runs was performed manually, whereas all of my implementations benefit from the ability to interface with a modern general-purpose CPU to establish which wheels to examine and when.

Overall, it is perhaps not surprising that, while high performance can be obtained using a general-purpose CPU just due to the massive improvements that have been made in transistor technology, the best performance arises from producing hardware to mimic how the problem would be solved mechanically. This is due to the exact extraction of parallelism that is possible with bespoke hardware but is necessarily missed at least partially by general purpose devices.

Personally, I have greatly enjoyed this project, both due to historical interest and to being able to explore the fields of General Purpose GPU programming and FPGA acceleration in more depth. The main difficulty I encountered was resolving the circular dependency between implementation and evaluation. For example, a lot of time was required to implement the algorithm on GPU and then test it, only to discover that the number of keys scored concurrently needed to be increased by at least an order of magnitude to utilise the GPU efficiently. In future projects I will schedule more time to

refine the implementations once an initial evaluation has been carried out.

5.2 Future Work

Codebreaking speed could be improved by:

- representing the entire key-scoring process as a vector operation;
- synthesising more cryptography modules onto the FPGA, allowing multiple wheels to be scored at once;
- synthesising the hardware design for an ASIC and fabricating a chip.

Bibliography

[1] *CUDA C Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2017-09-01.

[2] *CUDA C Programming Tutorials*. https://devblogs.nvidia.com. Accessed: 2017-09-01.

[3] *English Letter Frequency Counts: Mayzner Revisited*. http://norvig.com/mayzner.html. Accessed: 2018-04-13.

[4] Paul Gannon. *Colossus: Bletchley Park's Last Secret*. Atlantic Books Ltd, 2014.

[5] Jack Good, Donald Michie, and Geoffrey Timms. "General report on tunny". In: *National Archives/Public Records Office, HW* 25.4 (1945). Accessed: 2017-10-31.

[6] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[7] Kamran Karimi, Neil G Dickson, and Firas Hamze. "A performance comparison of CUDA and OpenCL". In: *arXiv preprint arXiv:1005.2581* (2010).

[8] *Memory Mapping for GPIO*. https://elinux.org/EBC_Exercise_11b_gpio_via_mmap. Accessed: 2018-05-07.

[9] *Optimizing Parallel Reduction in CUDA*. developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf. Accessed: 2018-01-12.

[10] *Schüth's SZ42 Codebreaking Software*. http://www.schlaupelz.de/SZ42/SZ42_software.html. Accessed: 2017-10-31.

[11] *Synchronization using PThreads*. https://github.com/angrave/SystemProgramming/wiki/Synchronization%2C-Part-8%3A-Ring-Buffer-Example. Accessed: 2017-12-17.

[12] *Terassic DE1-SoC Board Tools*. http://www.terasic.com.tw. Accessed: 2018-05-07.

Appendix A

Figures

1.1	Source: `https://en.wikipedia.org/wiki/File:Lorenz-SZ42-2.jpg`. License: Public Domain
2.1	Produced using draw.io.
2.2	Taken from Schüth's code [10] and cross-referenced with Gannon [4].
2.3	Frequencies computed by me on the first competition text. Chart produced in Microsoft Excel.
2.4	Chi-squared scores on *differenced* characters after decrypting with just the χ wheels. The ciphertext is the first competition plaintext encrypted with corresponding key. First three χ wheels initialised to $\chi_1 = 33, \chi_2 = 6$ (correct), $\chi_3 = 26$. Only the characters where *limitation* would not force ψ advance were used. Plot produced using Matlab.
3.2	gprof output of run on code compiled by GCC with optimisation level `-O0`. Run performed on first competition text. The profiling was performed on an early version of the algorithm before the high-level approach was finalised, so does not correspond exactly to any of the evaluated search methods.
3.3	Produced using draw.io.
3.4	Produced using draw.io.
3.5	Produced using draw.io.
3.7	Produced using draw.io.
3.8	Produced using draw.io.
3.9	Produced using draw.io.
3.10	Produced using draw.io.
3.11	Produced using draw.io.
4.1	Ada and CPU implementations run on a four-core Intel i5-3570K at 3.40 GHz; each compiled with maximum speed optimisation by Gnat and GCC respectively. GPU implementation run on NVIDIA GTX 760; compiled by CUDA with release settings. FPGA implementation run on Cyclone-V DE1-SoC, synthesised at 133.33 MHz by Quartus and using the Arm Cortex-A9 MPCore for coordination (HPS software compiled by Arm's C compiler). Error bars show 95% confidence interval of the mean. Graphs produced in Microsoft Excel.

4.4 Error bars show 95% confidence interval of the mean. Graph produced in Microsoft Excel.

4.5 Scoring performed on all wheels using consecutive positions from a randomised starting point (same seed for each version). The text used was the first "random words" text. All compilations performed by GCC, manually optimised versions compiled with `-O3` option. All scoring performed on Intel i7-6700HQ. Error bars show 95% confidence interval of the mean. Graph produced in Microsoft Excel.

4.7 Scoring performed only on χ wheels using consecutive wheel positions from zeroed starting point. First "random words" text used. Error bars show 95% confidence interval of the mean. Scoring performed on NVIDIA GTX 760. Graph produced in Microsoft Excel.

4.8 Scoring performed only on χ wheels using consecutive wheel positions from zeroed starting point. First "random words" text used. Error bars show 95% confidence interval of the mean. Scoring performed on NVIDIA GTX 760. Graph produced in Microsoft Excel.

4.9 Scoring performed on first "random words" text on NVIDIA GTX 760.

Appendix B

International Telegraph Alphabet No. 2

This is the code used to represent the plaintext characters before they are encrypted [5]. The code consists of two alphabets, which can be toggled between using the shift characters: < enables the figure alphabet and > enables the letter alphabet. Note that some of the interpretations of the figure alphabet varied.

Encoding	Letter	Note	Figure	Note
00000	i	Ignored in plaintext	i	Ignored in plaintext
00001	T		5	
00010	r	Carriage return	r	Carriage return
00011	O		9	
00100	_	Space	_	Space
00101	H		£	
00110	N		,	
00111	M		.	
01000	n	Newline	n	Newline
01001	L		)	
01010	R		4	
01011	G		@	
01100	I		8	
01101	P		0	
01110	C		:	
01111	V		=	
10000	E		3	
10001	Z		+	
10010	D		d	ID request
10011	B		?	
10100	S		’	
10101	Y		6	
10110	F		%	
10111	X		/	
11000	A		-	
11001	W		2	
11010	J		j	Ring bell
11011	<	Change to figure alphabet	<	Stay in figure alphabet
11100	U		7	
11101	Q		1	
11110	K		)	
11111	>	Stay in letter alphabet	>	Change to letter alphabet

Appendix C

Test texts

The plaintexts used for test runs throughout the project are divided up into the following four categories. Exactly the same code is run to decrypt in each case, i.e. the particular properties of the plaintext are not taken into account. Each text is encrypted with a distinct random key (both in terms of pattern and initial position) and all texts are encrypted with the χ-2 *limitation*.

C.1 Competition texts

The texts released on Schüth's website, used during the Bletchley Park competition. These consist of German words, but notably have artificially improved statistical characteristics, since every space character is surrounded on both sides by a pair of shift characters. This is unnecessary, since a space does not even require a shift, although the SZ42 operators would often double-up shift characters when they were used since one getting lost causes large chunks of text to become gibberish for the receiver.

There are three texts, of lengths 5,880, 5,880, and 5,850. The first hundred characters of the first text are as follows:

```
QZAHLEN<<,>><<_>>BITiE><_>V<<_((d83<<_>QBUM<-j?-43
<<_>>QELT<g8>>VON<<_>>NULL<<_>>BIS<<_>>UNENDLICH<<
```

C.2 Constant texts

This consists of the same character `A` repeated 6,000 times. The idea is to test the performance in the most favourable possible of cases.

C.3 Prose texts

These are texts designed to be representative of written English. They consist of punctuated continuous English writing, taken from Jane Austen's Pride And Prejudice (most commonly downloaded text on Project Gutenburg, `http://www.gutenberg.org/files/1342/1342-0.txt`, freely available under the license). One hundred 6,000 character sections were taken. Following is the first one hundred characters of the first text:

```
PRIDE_AND_PREJUDICEnnBY_JANE_AUSTENnnnnCHAPTER_1nn
nIT_IS_A_TRUTH_UNIVERSALLY_ACKNOWLEDGED_THAT_A_SIN
```

C.4 Random words texts

These are one hundred texts formed from concatenating random words taken from the dictionary in `/usr/share/dict/words` from Lubuntu. Apostrophised words are truncated from the apostrophe onwards to prevent `s` being overly common, and all special characters (i.e. non-alphabetic characters) are removed. The idea behind these texts is to be as uniform as possible while still being recognisably English. Each text is 6,000 characters long. The first hundred of the first text are as follows:

```
IMPRECISIONTARANTULASCATALYZINGCORNELIADISREGARDIN
GROSALIEVOUCHERSTALLEYRANDPINPRICKSBUNTSRESTORATIO
```

Appendix D

Project proposal

Peter Rugg
Churchill College
pdr32

Computer Science Tripos Part II Project Proposal

Parallel Codebreaking

12th October, 2017

Project Originator: Prof. Simon Moore

Project Supervisor: Prof. Simon Moore

Director of Studies: Dr John Fawcett

Overseers: Dr Rafał Mantiuk and Dr Ian Wassell

Introduction and Description of the Work

World War II was a key driver of cryptographic developments, due to the importance for all parties to communicate command information across large distances over easily intercepted radio transmissions. In particular, there was a race to develop sufficient pseudo-random stream generators to allow the ancient Vigenère cipher to be used practically over the large distances and with the high throughput required. One attempt at this was the German Lorenz cipher, introduced during the war for very high security messages. The strategic advantage for the Allies to being able to break this cipher was enormous, and so huge efforts were invested to applying cryptanalysis to the problem at the British codebreaking centre Bletchley Park. This eventually lead to the design and manufacture of several Colossus machines which automated certain parts of the deciphering process.

This project aims to investigate the technique used to break the Lorenz cipher (version SZ42): in particular the potential to exploit the parallelism present in the algorithm. The project will evaluate the performance on three different platforms: CPUs, GPUs, and (as an extension) FPGAs.

The SZ42 worked using a series of wheels, moving as a character was pressed on the attached keyboard. Cams (pins) on the wheels could be raised or lowered in advance by the operator, essentially determining the key: this determined how each wheel's movement affected that of the wheel next to it. The positions of the wheels then determined a pad to be XOR'ed with the entered character (expressed as a 5-bit code). The starting positions of the wheels formed the second part of the key, resulting in a total key combination count of approximately 10^{170}. Decryption could be performed symmetrically by starting the device with the same key settings as were used for encryption.

Cryptanalysts at Bletchley Park managed to deduce the workings of the device despite having never seen one. This allowed them to notice weaknesses in the pseudo-random generation which meant patterns persisted between plaintext and ciphertext when individual wheel settings were guessed correctly. This meant that a likely key could be built up one wheel at a time, making breaking the cipher feasible with some computational aid, provided by the Colossus machines. Joachim Schüth has implemented code to efficiently carry out the computations of Colossus (see starting point). This code will be used in the project to give a concrete implementation so the exact algorithm can be understood and adapted for the different platforms.

CPUs provide Multiple Instruction Multiple Data (MIMD) parallelism via the thread model. This will be used in the project to speed up the deciphering process, e.g. by trying different key settings in different threads. Care will need to be taken that synchronisation between the threads can occur without a large overhead.

GPUs provide Single Instruction Multiple Data (SIMD) parallelism, performing similar operations on many elements of a vector in parallel. This may be exploitable in the process of checking long ciphertext strings for patterns after being XOR'ed with some

stream determined by a candidate (sub-)key.

An FPGA (Field-Programmable Gate Array) can be used to model hardware which provides a mix of the above two forms of parallelism, tailored to the particular application. The ability to design custom application-specific pipelines allows for large performance benefits at the expense of a large design effort.

The overall motivation of the project is to see how a given workload can be adapted for different platforms. The particular choice of codebreaking as the application is mostly down to historical interest: modern cryptographic protocols are unlikely to be vulnerable to similar attacks due to the modern trend for proving security based on well studied intractable mathematical problems. However, many contemporary applications, e.g. in bioinformatics and malware detection, reduce down to a similar workload: searching long strings for patterns. Certainly the process of adapting algorithms to exploit parallel processing is becoming increasingly relevant as the limits of Moore's Law are reached and performance must be increased by increased core-count rather than increased clock speed. Hardware acceleration using FPGAs is also a currently active field as many high-performance applications, e.g. machine learning, require increased performance through heavy customisation.

Starting Point

The algorithm used will be adapted from the Ada implementation by Joachim Schüth for a Cipher Event competition available at `http://www.schlaupelz.de/SZ42/SZ42_software.html`. I do not know Ada apart from the brief introduction given in the IB Concurrent and Distributed systems course, so learning the language to a sufficient degree to understand Schüth's code will be allocated time in the timetable.

I have relatively high proficiency in the C programming language, having worked in C during a summer internship, including some work with threads but not synchronisation primitives.

I have no prior experience programming for GPUs beyond that covered in the IB Computer Design course. This will therefore need to be learned as part of the project.

I have more familiarity with FPGA programming, again from the IB Computer Design and ECAD and Architecture courses but also from a summer internship with the Computer Architecture Group at the DCST, which included the implementation of CPUs on FPGA using SystemVerilog. Interfacing between a CPU and FPGA for hardware acceleration is not something I have experience in, however, so this would have to be investigated for the project.

In terms of cryptanalysis, my experience is limited to the IB security course and some background research.

Substance and Structure of the Project

The project's objective is to produce a program which can perform a ciphertext-only attack on the SZ42 Lorenz cipher. This means accepting as an input a file containing an enciphered message and producing a file containing the original message. The user interface is not a priority for this project, so it is expected that the arguments will be supplied via the command line in the CPU and GPU implementations, and perhaps an SD card for the FPGA implementation. Signal processing is not an objective of the project, unlike Schüth's: the inputs will be taken in text form.

The CPU version should be multi-threaded and the focus of the evaluation will be multi-threaded performance, including how the performance scales to different numbers of cores. In particular, the metric will be running time on a single ciphertext (latency) rather than ciphertexts decoded per unit time (throughput). This is because the latter would reward just deciphering one ciphertext per thread, not leading to any interesting parallelism being extracted from the algorithm itself.

The GPU version should exploit the GPU's SIMD capabilities to improve performance: the focus of the evaluation in this case will be to what extent this has been achieved. Comparing the time spent in different stages of the process on the GPU vs on the CPU will also be a very interesting topic for evaluation.

The project naturally lends itself to being broken up into two components (plus one for extension): one for each different type of implementation. The CPU version is relatively low-risk, since the thread model is very familiar. The GPU version is higher risk (and effort) since a new model of concurrency has to be understood.

The project will approach each component using the waterfall software engineering approach: each component will be designed and planned carefully before implementation is started. This is reflected in the timetable, with a two week period allocated for planning before code is written for each component. This approach has the advantage of not requiring multiple iterations, which would use a lot of time in a project over a relatively short time period.

Extension

The main extension will be an FPGA version of the codebreaking algorithm. This will likely involve an integrated hard processor with FPGA acceleration (e.g. that provided by the DE1-SoC devices used in the IB ECAD and Architecture labs), with the CPU handling overall management but dispatching parallelisable parts of the computation to one of several FPGA pipelines. Implementing the pipelines themselves (most likely in SystemVerilog) is the focus of the extension, with a standard interface being used for interaction between the hard processor and FPGA if possible. Since an FPGA will have

a much lower clock speed than a CPU or GPU, this must be taken into account during evaluation, for instance by comparing performance per clock.

The FPGA version is a very high risk part of the project due to the large amount of design effort required in hardware design. This is why it has been suggested as an extension.

Depending on the time available, a variable amount of optimisation can be performed on the different implementations, such as adapting to better exploit memory hierarchies or more efficient pipelining of the FPGA implementation. In the very unlikely case all work is completed early, other cypher systems (e.g. Enigma) could be investigated in a similar way.

Success Criteria

The following minimum criteria define the success of the project:

1. A multi-threaded CPU application which can carry out a ciphertext-only attack on the SZ42 Lorenz cipher.
2. A GPU application which can carry out a ciphertext-only attack on the SZ42 Lorenz cipher. The application must exploit GPU parallelism.
3. Extension: an FPGA accelerator for carrying out a ciphertext-only attack on the SZ42 Lorenz cipher.
4. A quantitative evaluation of the effectiveness of the techniques.
5. A completed dissertation.

Risk Mitigation

The following table details my currently perceived risks associated with the project and what I plan to do to mitigate them:

Risk	Mitigation
Parallelising algorithm or GPU programming proves harder than expected	Making the FPGA implementation an extension means time allocated for this can be used for the core components instead if they take longer than anticipated.
CPU, GPU and FPGA implementations are implemented with time to spare	Open-ended extensions allow extra time to be spent improving implementations or performing other relevant work.
The algorithm used by Schüth relies on properties of the German plaintext not shared by English	In this case, machine translation (e.g. Google Translate) may need to be used to generate plaintexts for testing and evaluation. German corpora of suitable text could also be used. At the very least, the example ciphertexts used in the Cipher Event for which Schüth wrote the original code are available.
Data loss	Suitable backup protocols will be used (see Resource Declaration).

Timetable and Milestones

Following is the timetable of the project, split into two week periods starting on the Thursday the day before the formal proposal deadline and ending the Wednesday two days before the dissertation deadline. The timetable shows the weeks during which lectures are taking place: it is expected that less time will be devoted to the project during these weeks due to time spent attending lectures and completing supervision work.

The timetabled aim is to complete the core implementation goals by the end of the Christmas vacation, allowing detailed performance analysis and evaluation during Lent term when the DCST computing resources can more easily be accessed. In addition, if progress has been made sufficiently quickly, this makes the FPGA extension a very realistic possibility. On the other hand, this leaves plenty of contingency (time labelled as slack) in case progress is slower than expected. The aim is to complete a draft of the dissertation towards the start of the Easter vacation to allow focus on revision, with only incorporation of corrections and minor modifications taking place after this. The final dissertation will be submitted at least a week before the final deadline. The weeks towards the end of the Easter vacation and during Easter term are included in the timetable to give a sense of

the available time in case of overrunning, even though the plan is that the project and dissertation will be nearly completed by this point.

Note that testing of software is implicit in implementation, so not listed as separate work.

19th October, 2017-1st November, 2017

Scheduling Considerations

- Proposal deadline: 20th October, 2017.
- Full lecture timetable.

Work

- Background reading motivated by producing an initial draft of the Introduction of the dissertation. This will likely have to be changed later as relevant points to emphasise become clearer.
- Implementation of a C version of the SZ42 cipher/decipher machine: while existing code exists to encipher and decipher texts, this will help with understanding the machine and produce a version which can easily interoperate with later work.

Milestones

- Working implementation of SZ42.
- Draft of Introduction chapter.

2nd November, 2017-15th November, 2017

Scheduling Considerations

- Full lecture timetable.

Work

- Investigation of the approach taken for breaking the SZ42 cipher. This will involve familiarising myself with Ada to understand and annotate Schüth's code.
- Planning on how to parallelise the algorithm.

Milestones

- Fully annotated version of the Ada code.
- Document describing planned approach.

16th November, 2017-29th November, 2017

Scheduling Considerations

- Full lecture timetable.

Work

- Parallel C implementation of codebreaking algorithm.

Milestones

- Working C implementation.

30th November, 2017-13th December, 2017

Scheduling Considerations

- No lectures (Christmas vacation).

Work

- Planning for experiments to carry out to evaluate performance of implementations.
- Slack/extension: initial research into hardware acceleration, including interfaces between hard processor and FPGA.

Milestone

- Document describing experiments to be carried out for Evaluation.
- Extension: "Hello world" FPGA acceleration project.

14th December, 2017-27th December, 2017

Scheduling Considerations

- No lectures (Christmas vacation).
- Somewhat lightened workload on week of Christmas.

Work

- Investigation of GPU programming and planning approach for adapting codebreaking algorithm.

Milestones

- Document describing approach to exploit SIMD parallelism.

28th December, 2017-10th January, 2018

Scheduling Considerations

- No lectures (Christmas vacation).

Work

- GPU implementation of codebreaking algorithm.

Milestones

- Working GPU implementation.

11th January, 2018-24th January, 2018

Scheduling Considerations

- Lectures begin: 18th January, 2018.

Work

- Evaluation of CPU and GPU version performance.

Milestones

- Data collected detailing relative performance of components.

25th January, 2018-7th February, 2018

Scheduling Considerations

- Progress report deadline: 2nd February, 2018.
- Full lecture timetable.

Work

- Work on progress report and presentation.
- Work on Preparation and Implementation (but not of extension) sections of the dissertation.
- Extension: design FPGA pipelines for implementing the algorithm.

Milestones

- Progress report document.
- Progress report slides and practised presentation.
- Drafts of Preparation and Implementation chapters.
- Extension: diagram describing high level design of FPGA pipeline.

8th February, 2018-21st February, 2018

Scheduling Considerations

- Progress report presentations: 8th February, 2018-13th February, 2018.
- Full lecture timetable.

Work

- Slack/extension: implementation of FPGA accelerator

Milestones

- Extension: working FPGA implementation.

22nd February, 2018-7th March, 2018

Scheduling Considerations

- Full lecture timetable.

Work

- Slack/extension: evaluation of FPGA performance.

Milestones

- Extension: data collected about FPGA performance.

8th March, 2018-21st March, 2018

Scheduling Considerations

- Lectures end: 14th March, 2018 (Easter vacation).

Work

- Work on completion of dissertation draft, compiling results of comparison into the Evaluation and Conclusions sections and finalising other sections with overall project findings in mind.
- Compilation of data into graphs and production of relevant diagrams.

Milestones

- Completed dissertation draft submitted to supervisor.

22nd March, 2018-4th April, 2018

Scheduling Considerations

- Exam revision.
- No lectures (Easter vacation).

5th April, 2018-18th April, 2018

Scheduling Considerations

- Exam revision.
- No lectures (Easter vacation).

19th April, 2018-2nd May, 2018

Scheduling Considerations

- Exam revision.
- Reduced lectures begin: 26th April, 2018.

Work

- Incorporation of supervisor's and friends' dissertation corrections.
- Final proofreading of dissertation.

Milestones

- Completed dissertation.

3rd May, 2018-16th May, 2018

Scheduling Considerations

- Dissertation deadline: 18th May, 2018.
- Exam revision.
- Reduced lecture timetable.

Resources Declaration

Hardware

For the majority of the project, my own desktop machine (3.4GHz Intel i5 CPU with 16GiB of RAM and GTX 760 graphics card) will be used for convenience, with my laptop as backup (2.6Ghz Intel i7 CPU with 16GiB of RAM and GTX 960M graphics card). The similarity of the two systems should allow relatively easy transition if one fails, even for GPU programming. I accept full responsibility for these machines and I have made contingency plans to protect myself against hardware and/or software failure.

Access to computer architecture group servers is also requested in order to access FPGAs and a wider variety of high-performance GPUs to test on: Prof. Simon Moore has kindly agreed to take responsibility for this access and is listed on the cover sheet to this effect.

Software

A GPU compiler will be required: the exact software will be determined during the project based on research, but the two most widespread options (CUDA and OpenCL) are freely available so it seems unlikely paid-for software will be required.

Similarly, an Ada compiler may be required to investigate Schüth's original code. GNAT is freely available for this purpose.

Schüth's code itself has been made available for people to use and learn from, so this project's use falls within its intended purpose. A backup of the files has already been made in case the page becomes unavailable during the project.

For the FPGA programming, software from the IB ECAD and Architecture labs for FPGA simulation (e.g. ModelSim) and synthesis (e.g. Quartus) would be required. These are already available on the MCS for the IB practicals.

Backup Provision

To protect against data loss, I will use the git version control system for the dissertation and all software, storing the repository on GitHub, a local hard drive automatically backed up to Google Drive, and regularly backed up to an external hard drive. For further protection, I will make a weekly copy of the repository to a memory stick used only for this purpose. By making regular commits I am protected against accidental overwrite of my own work, and with the many forms of redundancy I am protected against external factors such as a data loss by Google, disk failure, or ransomware attack.